Lisp

Eine Einführung

Sven Naumann

Wintersemester 2007

Inhaltsverzeichnis

Kapitel 1

LISP-Geschichte(n) & Grundlagen

1.1 Kurze Darstellung des Lambda-Kalküls

1.1.1 Historische Hintergründe

Alonzo Church entwickelte zu Beginn des 20. Jahrhunderts, als mit unterschiedlichen Ansätzen versucht wurde, die Grundlagenkrise der Mathematik zu überwinden (*Logizismus* [Frege, Russel]; *Intuitionismus* [Brouwer]; *Beweistheorie* [Hilbert]), ein aussagenlogisches System, in dem der *Lambda-Operator* eine zentrale Rolle spielte. Auf der Grundlage dieses Systems, von dem er hoffte, es könne zur Lösung der Probleme der naiven Mengenlehre verwendet werden, entwickelte er später den ***Lambda-Kalkül*** , der wesentlich zur Präzisierung des Funktionsbegriffs, sowie der Entwicklung der Theorie der rekursiven Funktionen beitrug.

Der Lambda-Kalkül wurde dann von John McCarthy bei der Entwicklung der Programmiersprache LISP als Mittel zur Notierung von Funktionen und zur Klärung von Bindungsprozessen der beteiligten Variablen verwendet[1].

[1] Ein *klassisches* Bindungsproblem ist das sogenannte **FUNARG**–Problem: Wie sind die freien Variablen einer Funktion f_1 zu binden, die selbst als Argument (**fun***ctional* **arg***ument*) einer anderen Funktion f_2 verwendet wird? Sind sie innerhalb des Kontextes bzw. der Umgebung zu interpretieren, innerhalb dessen f_1 definiert wurde oder innerhalb der Umgebung, in der f_1 verwendet wird?

1.1.2 Kurze Darstellung der Grundzüge des Lambda-Kalküls

Eine Funktion kann als eine Zuordnungsvorschrift aufgefaßt werden, die Elemente aus ihrem Definitionsbereich auf Elemente aus ihrem Wertebereich abbildet. Mengentheoretisch läßt sich daher eine beliebige Funktion f mit einer Menge von Paaren <x, y> identifizieren, für die gilt: x ist ein Element des Definitionsbereiches von f, y ist ein Element des Wertebereiches von f, und f bildet x auf y ab (f(x) = y). Diese Notation für Funktionen ist allerdings sehr umständlich und für Funktionen mit nicht-endlichem Definitions-/Wertebereich kaum realisierbar. Eine Alternative bildet in vielen Fällen die Darstellung einer Funktion durch eine Berechnungsvorschrift.

Beispiel (1-1)
Berechnungsvorschrift: $(x^2 + x)^2$
Instanzen: $1 \longrightarrow 4,\ 2 \longrightarrow 36, \ldots$

Um einen Ausdruck dieses Typs zur Bezeichnung einer Funktion verwenden zu können, ist es notwendig, x als Variable zu kennzeichnen und zu binden; d.h. es muß herausgestellt werden, daß für x beliebige Elemente des Definitionsbereiches der zu bezeichnenden Funktion *substituiert* werden können und daß bei einer derartigen Substitution für beide Vorkommen von x derselbe Wert zu substituieren ist. Church schlug vor, zu diesem Zweck einen Operator einzuführen, zu dessen Bezeichnung er das griechische Zeichen Lambda verwendete (daher die Bezeichnung *Lambda-Operator*). Ausgehend von dem letzten Beispiel erhält man den Funktionsbezeichner (λx $(x^2 + x)^2$). Allgemein gilt:

> *Wenn* E *ein Ausdruck ist, der* x *als freie Variable enthält, dann bezeichnet (*λ xE*) die Funktion, die für ein Argument* a *aus dem Definitionsbereich den Wert liefert, der sich nach der Substitution von* a *für* x *in* E *durch die Ausführung der Berechnungsvorschrift ergibt.*

Da Church den Prozeß der Bildung von Funktionszeichen aus Berechnungsvorschriften als Abstraktion betrachtet, bezeichnet der den Lambda-Operator auch als *Abstraktionsoperator*.

1.1.3 Einige Charakteristika des Lambda-Kalküls

- Es gibt zwei Typen von Ausdrücken:
  - Funktionsausdrücke der Form ($\lambda x\ E$) und
  - Funktionsanwendungen der Form ($F\ A$), wobei F die angewandte Funktion und A das Argument dieser Funktion bezeichnet.
- Mehrstellige Funktionen werden auf einstellige Funktionen zurückgeführt.

- Die Struktur des Lambda-Kalküls ist sehr einfach:
  Es gibt drei Kompositions- und drei Auswertungsregeln (*Konversionsregeln* genannt).
- Die Reihenfolge, in der komplexe Ausdrücke ausgewertet werden, ist irrelevant.
- Es ist nicht möglich, Seiteneffekte zu erzielen.

1.2 Entwicklung von LISP

LISP wurde Ende der 50er Jahre von John McCarthy entwickelt und ist damit zusammen mit FORTRAN eine der ältesten höheren Programmiersprachen Ihre Entwicklung ist eng mit den ersten Schritten der Disziplin verknüpft, die uns heute unter dem Namen 'Artificial Intelligence' vertraut ist:
1956 fand am Dartmouth College in New Hampshire die "Dartmouth Summer School on Artificial Intelligence" statt, die erste derartige Veranstaltung überhaupt, die von McCarthy mitorganisiert wurde. An ihr nahmen u.a. A.Newell und H.Simon teil, die im Rahmen der Entwicklung eines Theorembeweisers, der sogen. *Logic Theory Machine*, die Programmiersprache IPL (*Information Processing Language*) entwickelt hatten, deren herausragendste Eigenschaft die Verwendung von Listen als rekursiver Datenstruktur war.
Neben IPL V beeinflusste auch FORTRAN stark die Entwicklung der ersten LISP-Versionen. Die Möglichkeiten, durch Verschachtelung komplexe Funktionsausdrücke zu bilden und die Sprache durch selbstdefinierte Funktionen zu erweitern, waren für die damalige Zeit geradezu revolutionär. Allein die damals verbreitete Annahme, die *Artificial Intelligence* habe es vor allem mit der Verarbeitung symbolischer Ausdrücke und nicht mathematischer Terme zu tun, ließ FORTRAN als Sprache für diesen Bereich als wenig geeignet erscheinen. Gesucht wurde also eine Sprache zur Symbolverarbeitung mit Listen als rekursiver Datenstruktur und der Möglichkeit zur Bildung komplexer funktionaler Ausdrücke und Definition neuer Funktionen durch den Benutzer.

1958, als am MIT das *Artificial Intelligence Project* gestartet wird, an dem neben McCarthy auch M.Minsky arbeitet, beginnt McCarthys Arbeitsgruppe mit der Realisierung von LISP1. Zuvor war bei IBM eine FORTRAN-Version entwickelt worden, die durch Listenverarbeitungsfunktionen erweitert wurde, über einen experimentellen Status aber nicht hinauskam.

1.2.1 Die ersten Versionen

LISP 1 wird 1959 fertiggestellt. In dieser Phase orientiert sich LISP noch eng an FORTRAN, weist aber bereits eine Reihe nicht zu unterschätzender Vorteile auf:

- Es ist möglich, rekursive Funktionen zu definieren.

- Es wird eine dynamische Speicherverwaltung (*garbage collection*) verwendet.
- Ausdrücke werden in der syntaktisch eindeutigen Präfix-Notation (*polnische Notation*) notiert.

Allerdings gibt es auch eine Reihe von erheblichen Restriktionen: LISP 1 kennt keine ganzen (Typ: Integer), sondern nur Gleitkommazahlen und verwendet nicht die LAMBDA-Notation zur Bezeichnung von Funktionen. Auch steht zunächst kein Compiler, sondern nur ein Interpreter zur Verfügung. Die Sprache umfaßt 90 vordefinierte Funktionen (zum Vergleich: *Common LISP* unfasst etwa 800 Funktionen) und wird anfangs auf einer IBM 704 implementiert.

LISP 1.5 wird 1962 am MIT fertiggestellt. Die wichtigsten Verbesserungen gegenüber LISP 1 lassen sich wie folgt zusammenfassen:

- Neben reelen gibt es jetzt auch den Datentyp der ganzen Zahlen.
- Der für LISP so charakteristische Typ der *Paarlisten* (CONSes) wird eingeführt.
- Viele Funktionen erlauben die Verwendung einer beliebigen Zahl von Parametern.
- Die Ein-/Ausgabefunktionen und arithmetische Funktionen werden erheblich verbessert.

Der Sprachkern besteht aus 150 Funktionen. Das System ist auf einer IBM 7090 implementiert.

1.2.2 Die lokale Verbreitung von LISP (1962-1966)

Durch McCarthys Wechsel vom MIT zum Stanford College etabliert sich dort ab 1962 ein zweites LISP-Zentrum. Ein weiteres entsteht bei der Datenverarbeitungsfirma BBN (Bolt, Beranek & Newman) unter der Leitung von L.S.Bobrow. Besonders diese drei Zentren treiben die Weiterentwicklung von LISP in diesen Jahren voran, entwickeln neue LISP-Versionen und besonders verbesserte Entwicklungsumgebungen mit leistungsfähigen Debugging-Möglichkeiten und Strukturediteren, die die Erstellung von LISP-Programmen unterstützen.

> **Zeithistorische Randnotiz**
>
> *Bei einer 1964 von BBN durchgeführten empirischen Studie zur 'Qualität' von Programmiersprachen schneidet LISP erstaunlich gut ab: Bei dieser Untersuchung wurden 12 Programmierer, die für eine der zu dieser Zeit verbreiteten Programmiersprachen als Experten galten, mit dem Problem konfrontiert, für sieben Aufgaben aus verschiedenen Wissensbereichen Programme zu entwickeln. Anschließend wurden die benötigte Zeit und die erstellten Programme miteinander verglichen: Die Bearbeitung der Aufgaben erforderte mit FORTRAN 16*

Stunden, mit ALGOL 7 Tage und mit COBOL sogar 5 Wochen. Die LISP-Programme dagegen waren schon nach 4 Stunden fertiggestellt.

1.2.3 Die weltweite Verbreitung von LISP nach 1966 - Dialektologie

Die weltweite Verbreitung von LISP in den darauffolgenden Jahren, die nicht unwesentlich durch den von Bobrow und Berkeley herausgegebenen Sammelband "The Programming Language LISP: Its Operation and Applications" und die IFIP Konferenz 1966 in Pisa über symbolverarbeitende Sprachen gefördert wird, ist durch zwei Tendenzen gekennzeichnet:

1. LISP wird auf immer neuen Rechnertypen implementiert und in neue Umgebungen eingebunden.

2. Es verstärkt sich die Tendenz zur Entwicklung neuer LISP-Dialekte mit z.T. sehr unterschiedlichen Merkmalen.

Mit der Entwicklung von Common LISP ab 1980 wird versucht, einen Sprachstandard zu setzen, der sich dann auch in den nächsten Jahren - vor allem in den USA - weitgehend durchsetzt und schließlich auch in Europa und Japan immer größere Akzeptanz gewinnt. Ende der 80er Jahre beginnen eine amerikanische ANSI-Kommission, die europäische EuLISP-Kommission, eine französische AFNOR-Kommission und eine deutsche DIN-Kommission Vorschläge zu einer weiteren Standardisierung von LISP auf der Basis von Common LISP[2] zu erarbeiten, die schließlich im Dezember 1994 zu der Verabschiedung eines ANSI-LISP Standards führen. Es ist zu erwarten, daß dieser Standard in den kommenden Jahren die Grundlage für künftige LISP-Implementierungen bilden wird. **Der**

LISP-Stammbaum

[2]Das unbestrittene Standard-Nachschlagewerk für Common LISP, das aufgrund seines Umfangs allerdings nur von denen sinnvoll genutzt werden kann, die schon über grundlegende LISP-Kenntnisse vefügen, ist:
G.L.Steele *Common LISP. The Language.*, Digital Press, 2. Aufl. 1990.

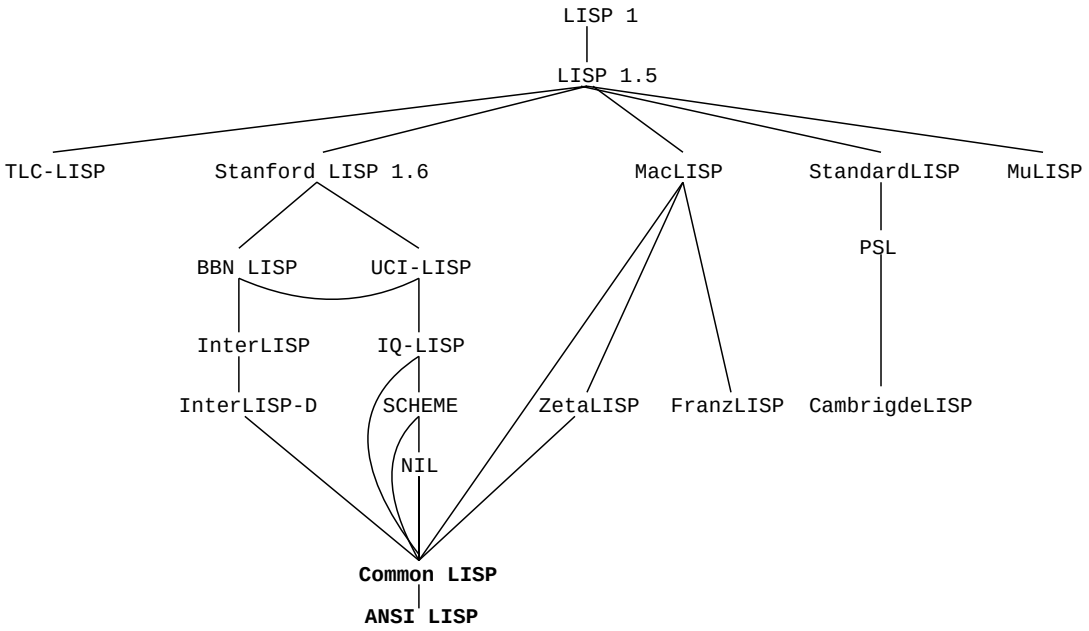


Abbildung (1-1)

1.3 LISP-Systeme

1.3.1 LISP-Programmierumgebungen

Aufbau. Eine Programmierumgebung besteht aus einer Gruppe von Modulen zur Entwicklung und Ausführung von Programmen:

Kern einer LISP-Programmierumgebung bildet der LISP-Interpreter, der als ein interaktives Werkzeug eine schnelle Entwicklung und Überprüfung kleiner Programme erlaubt. Neben dem Interpreter stehen in der Regel auch ein (Struktur-)Editor, ein Compiler und ein Modul mit Programmen zur Fehlersuche (Debugger: *trace*, *backtrace, step*, *debug*) zur Verfügung. Häufig sind einige dieser Module, z.B. die Programme zur Fehlersuche, in LISP selbst implementiert.

Der LISP-Interpreter. Aufgrund seiner Arbeitsweise wird der LISP-Interpreter häufig als *read-eval-print*-Schleife bezeichnet; denn solange der Interpreter aktiv ist, wird immer wieder die folgende Operationssequenz ausgeführt:

1. Zunächst wird ein beliebiger symbolischer Ausdruck erwartet und eingelesen (*read*).
2. Wenn dieser Ausdruck syntaktisch wohlgeformt und auswertbar ist, dann wird der Wert dieses Ausdrucks berechnet (*eval*)
3. Der Wert des Ausdrucks bzw. eine Fehlermeldung wird ausgegeben (*print*).

Tritt ein Fehler auf, wird ein sogen. *break-level* aktiviert, was dem Benutzer in der Regel durch eine Zahl ($\geq$1), die vor dem Prompt erscheint, signalisiert wird. Innerhalb des *break-levels* hat er Zugriff auf die aktuelle Bindungsumgebung und so die Möglichkeit, das Programm direkt zu modifizieren und Fehler schnell zu beseitigen.

1.3.2 Speicherverwaltung

Segmentierung des Speichers. In vielen Fällen wird der dem LISP-System zur Verfügung stehende Speicherbereich in drei Hauptbereiche zerlegt. So gibt es:

(i). einen Bereich, in dem die Werte für temporäre Variablen und Rücksprungadressen gespeichert werden;

(ii). einen Bereich, in dem der binäre Programmkode abgelegt wird und

(iii). einen Bereich, in dem alle LISP-Objekte wie Symbole, Listen, etc. verwaltet werden.

Verwaltung der Objekte. In vielen Systemen werden die Objekte über eine Objektliste mit der Bezeichnung $*oblist*$ oder $*obarray*$ verwaltet. Diese Objektliste enthält allerdings nicht die Objekte, sondern Zeiger auf diese Objekte; d.h. die Referenz auf Objekte erfolgt immer *indirekt*: ein Symbol, das ein LISP-Objekt bezeichnet, verweist auf ein Feld der Objektliste und der in diesem Feld enthaltene Zeiger auf das Objekt.

Ein Vorteil dieses Verfahrens liegt darin, daß, um zu überprüfen, ob zwei Symbole dasselbe Objekt bezeichnen, nicht die möglicherweise recht komplexen Objekte miteinander verglichen werden müssen, sondern nur zu prüfen ist, ob sie auf dasselbe Feld der Objektliste verweisen:

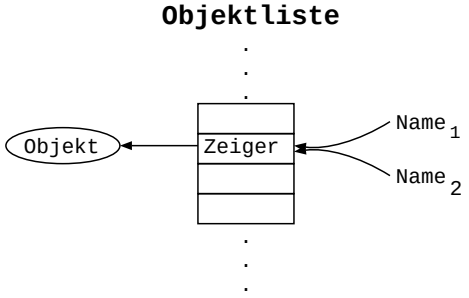


Abbildung (1-2)

Um Angaben über den Typ der in diesem Bereich verwalteten Objekte repräsentieren zu können, wird entweder der Objekt-Speicherbereich weiter segmentiert (ein Bereich für Symbole, ein Bereich für Listen, ...) oder durch Type-Bits innerhalb des Objekts selbst kodiert.

Garbage Collection. Der Speicherplatz für komplexe Objekte wird in LISP über eine *free-storage list* verwaltet. Da bei der Ausführung eines LISP-Programms ständig neue Objekte

erzeugt werden können, kann die Grenze des zur Verfügung stehenden Speicherplatzes schnell erreicht werden. Wenn der Speicherplatz knapp zu werden droht, wird der Speicher nach Objekten durchsucht, auf die zu diesem Zeitpunkt kein Zeiger mehr verweist, und der von ihnen beanspruchte Speicherplatz wird wieder der *free-storage list* zur Verfügung gestellt. Dieser Prozeß wird als "garbage collection" bezeichnet[3].

1.4 Datentypen

Common LISP kennt eine Vielzahl von vordefinierten Datentypen. Zu den wichtigsten gehören die folgenden:

1. **Atome**
   (a) selbstevaluierende Objekte
      - Zahlen
      - Zeichen
      - Zeichenketten

   (b) Symbole
   (c) Funktionen
   (d) Arrays
   (e) Strukturen
   (f) Streams
2. **Conses**
   (a) Paarlisten
   (b) Listen
      - einfache Listen
      - Assoziationslisten
      - Property-Listen

Es wird gelegentlich behauptet, ein entscheidender Unterschied zwischen LISP und einer Sprache wie PASCAL läge in der von PASCAL geforderten *strikten Typisierung*. Diese Behauptung ist nicht korrekt. Der entscheidende Unterschied zwischen diesen Sprachen liegt nicht in der Strenge der Typisierung, sondern darin, was typisiert wird:

[3] Eine anschauliche Beschreibung verschiedener Strategien, nach denen eine *garbage collection* ausgeführt werden kann, findet sich in: Winston & Horn (1989), S. 249-58.

In PASCAL werden **Variablen** typisiert, in LISP dagegen **Objekte**. Da in PASCAL Variablen nicht wie LISP-Variablen Namen von Objekten, sondern Namen von Speicherbereichen sind, ist es notwendig, dem Compiler für jede Variable per Variablendeklaration mitzuteilen, wie groß dieser Speicherbereich sein muß. In LISP sind solche Deklarationen nicht erforderlich.

Beispiel (1-2)
Objektrepräsentationen:

a) ganze Zahlen:
   1 -37466 0

b) Dezimalzahlen:
   0.0 3.14 6.02+E23 -0.0000000000009

c) Zeichen:
   #\a #\Z #\9 #\(#\= #\Newline

d) Zeichenketten:
   "Hallo" "" "Noch ein String" "\""

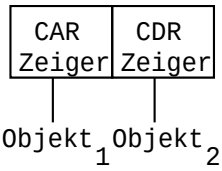
e) Symbole:
   HALLO Hallo mein-symbol datei1.lsp

f) Listen:
   () (eine liste) (noch (eine (Liste))) (eine . LISTE?)

Bevor wir die grundlegenden semantischen Regeln einführen, die die Interpretation von LISP-Ausdrücken und Programmen bestimmen, werfen wir zunächst noch einen kurzen Blick auf den für LISP charakteristischen Datentyp der CONSes.

1.5 CONSes

Ein CONS ist ein komplexes Objekt, das durch die geordnete Verbindung zweier LISP-Objekte gebildet wird, von denen jedes selbst wieder ein Objekt beliebigen Typs (z.B ein *CONS*) sein kann; d.h., CONSes bilden eine rekursive Datenstruktur. Objekte dieses Typs werden häufig durch sogen. *CONS-Zellen* repräsentiert, die zwei Zeiger enthalten, wobei der erste auf das erste Objekt und der zweite auf das zweite Objekt weist[4]:



Es gibt zwei Typen von CONSes: Listen und Paarlisten (*dotted pairs*). Der Unterschied zwischen diesen beiden Typen von CONSes besteht darin, daß bei Listen der CDR-Zeiger der letzten CONS-Zelle immer auf das die leere Liste repräsentierende Symbol NIL weist; bei den Paarlisten dagegen auf ein beliebiges LISP-Objekt. Listen bilden also einen speziellen Subtyp von Paarlisten - es sind Paarlisten, bei denen der CDR-Zeiger der letzten CONS-Zelle auf NIL weist.

Um für den Benutzer die Unterscheidung von Listen und Paarlisten transparent zu gestalten, werden Paarlisten vom System auf dem Bildschirm anders dargestellt als 'echte' Listen: Bei der Repräsentation von Paarlisten wird der Punkt "." verwendet, um die beiden Objekte der Paarliste voneinander zu trennen; d.h. der Punkt "." trennt den CAR-Teil vom CDR-Teil einer CONS-Zelle.

Beispiel (1-3)

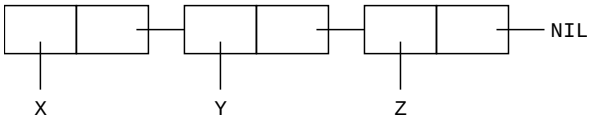
```
(X . Y)       ; Paarliste, die aus den Objekten X und Y besteht
(X Y)         ; Liste, die auch diese Objekte enthält
(X . (Y . Z)) ; Paarliste, die aus X und der Paarliste (Y . Z) besteht
(X Y Z)       ; Liste mit den Objekten X, Y, Z
```

Tatsächlich lassen sich auch Listen in dieser Punktnotation darstellen:

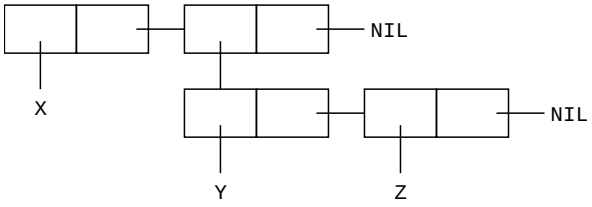
```
(X . (Y . NIL))
(X . (Y . (Z . NIL)))
```

Allerdings wird diese Repräsentation vom System direkt in die Standardlistenrepräsentation ohne Punkt überführt. Durch Verwendung von CONS-Zellen läßt sich die Struktur von Listen und Paarlisten sehr anschaulich darstellen; die Liste (X Y Z) z.B. läßt sich so repräsentieren als:

[4]Die Bezeichnungen *CAR-Zeiger* bzw. *CDR-Zeiger* sind historischen Ursprungs: *CAR/CDR* sind Abkürzungen für die Namen der entsprechenden Register der IBM 704, auf der LISP ursprünglich implementiert wurde.



Die Liste besteht in dieser Repräsentationsform aus drei CONS-Zellen: Der CAR-Zeiger der ersten weist auf das Objekt X, der CDR-Zeiger auf ein CONS, das aus den CONS-Zellen 2 und 3 besteht. Der CAR-Zeiger der zweiten CONS-Zelle weist auf das Objekt Y, der CDR-Zeiger auf ein CONS, das aus der CONS-Zelle 3 besteht, und der CAR-Zeiger der dritten CONS-Zelle weist auf das Objekt Z, der CDR-Zeiger auf NIL. Dieser Punkt ist, wie schon erwähnt, entscheidend: Der CDR-Zeiger der letzten CONS-Zelle einer Liste weist immer auf NIL. So ergibt sich z.B. für die Liste (X (Y Z)) folgende Struktur:



Zur Erzeugung von CONSes gibt es in LISP die Funktion **CONS** (*construct*)[5]:

CONS	*Objekt*$_1$ *Objekt*$_2$	[Funktion]
CONS erzeugt ein komplexes Objekt mit *Objekt*$_1$ als erster und *Objekt*$_2$ als zweiter Komponente.		

Beispiel (1-4)

(CONS 'A ())	$\Longrightarrow$[6]	(A)
(CONS 'A '(B))	$\Longrightarrow$	(A B)
(CONS 'A '((B)))	$\Longrightarrow$	(A (B))
(CONS '(A) '(B))	$\Longrightarrow$	((A) B)
(CONS () ())	$\Longrightarrow$	(NIL)

Die Listen (X Y Z) und (X (Y Z)) kann man - statt sie durch eine Terminaleingabe direkt zu *notieren* - durch folgende Anweisungen *generieren*:

(CONS 'X (CONS 'Y (CONS 'Z ())))
bzw.
(CONS 'X (CONS (CONS 'Y (CONS 'Z ())) ()))

[5] Alle LISP-Funktionen werden bei ihrer Einführung in diesem Format spezifiziert: Auf den Funktionsnamen folgt die Spezifizierung ihrer Parameter und eine Typbeschreibung in eckigen Klammern

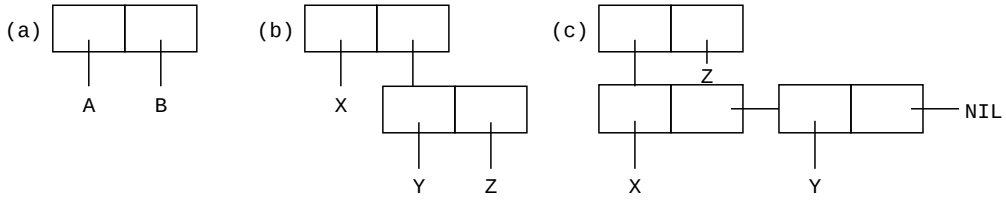
[6] "$\Longrightarrow$" ist ein metasprachliches Symbol, das zu lesen ist als: "evaluiert zu".

Natürlich lassen sich durch CONS nicht nur Listen, sondern auch Paarlisten generieren:

Beispiel (1-5)

```
(CONS 'A 'B)              ⟹   (A . B)
(CONS 'X (CONS 'Y 'Z))    ⟹   (X . (Y . Z))
(CONS '(X Y) 'Z)          ⟹   ((X Y) . Z))
                               ; eine Paarliste, die als erste
                               ; Komponente eine Liste enthält!
```

Als CONS-Zellen notiert:



Obwohl Paarlisten den fundamentaleren Objekttyp bilden, ist für die LISP-Programmierung in vielen Fällen die Verwendung von Listen vorzuziehen und der Rückgriff auf Paarlisten nur in bestimmten Situationen sinnvoll bzw. erforderlich. Manche LISP-ProgrammiererInnen lehnen die Verwendung von Paarlisten grundsätzlich ab[7].

1.6 Evaluierung von S-Expression

LISP-Datenstrukturen werden als *S-Expressions* (*symbolic expressions*) bezeichnet. Es sind nur eine geringe Zahl einfacher semantischer Regeln erforderlich um festzulegen, wie eine S-Expression zu interpretieren (*evaluieren*) ist.

1.6.1 Selbstevaluierende Objekte

Zahlen, Zeichen und Zeichenketten (*strings*) werden als *selbstevaluierende* Objekte bezeichnet. Die Evaluierung von S-Expressions dieses Typs, wird durch folgende Regel bestimmt:

Regel 1
Zahlen, Zeichen und Zeichenketten evaluieren zu sich selbst.

[7] Auf zwei spezielle Typen von Listen, **Assoziationslisten** und **Property-Listen**, werden wir in einer späteren Sitzung zu sprechen kommen.

Beispiel (1-6)

3	$\Longrightarrow$	3
#\a	$\Longrightarrow$	#\a
-4.7009-E11	$\Longrightarrow$	-4.7009-E11
"Ein alter Hut"	$\Longrightarrow$	"Ein alter Hut"

1.6.2 Symbole

LISP-Symbole sind komplexe Objekte, die in Common LISP Implementationen aus mindestens den folgenden fünf Teilen bestehen:

Symbol

print-name
value-binding
function-binding
property-list
package

] *Print-Name*. Der *print-name* des Symbols legt den Namen fest, den das System zur Verwaltung des Symbols verwendet. Da in Common LISP bei der Bildung von Symbolnamen nicht zwischen Groß- und Kleinbuchstaben unterschieden wird (Kleinbuchstaben werden automatisch in Großbuchstaben konvertiert), besteht der *print-name* eines Symbols in der Regel aus Großbuchstaben und anderen in Symbolnamen zulässigen Zeichen[8].

Beispiel (1-7)
TEILCHENBESCHLEUNIGER
TEilchenBESCHLeunIGER
teilchenbeschleuniger

Drei verschiedene Möglichkeiten, das Symbol mit dem print-name *TEILCHENBESCHLEUNIGER zu bezeichnen.*

∗globale-variable∗
ein-datei.name
ein/pfad/name

[8] Bei der Bildung von Symbolnamen sind neben Klein- und Großbuchstaben und den Zahlzeichen auch die folgenden Sonderzeichen zulässig: + - ∗ / @ $ % ^ & _ \ < > .
Die folgenden Zeichen sind zwar auch grundsätzlich zulässig; allerdings sind sie für die Definition von READ-Makros durch den Benutzer reserviert: ? ! [ ] { }

Drei Symbole, deren print-name *Sonderzeichen enthält.*

Symbolen können Werte zugewiesen werden, die abhängig von der Art des Wertes und der Form der Wertzuweisung in dem mit *value-binding*, *function-binding* (im folgenden auch: *Wertbindung* bzw. *Funktionsbindung*) bzw. *property-list* bezeichneten Teil des Symbols gespeichert werden[9]. Wir werden uns zunächst auf die Wert- und Funktionsbindung von Symbolen beschränken.

Wertbindung. Da es möglich ist, Symbolen Werte zuzuweisen, können sie als Variablen verwendet werden. Als Werte für Symbole sind beliebige LISP-Objekte zulässig. So kann z.B. der Wert eines Symbols ein anderes Symbol sein.

Beispiel (1-8)

print-name	*value-binding*
APFEL	BANANE
KIRSCHE	KIRSCHE
PI	3.14159
NAME	"Otto Mustermann"
HEIDI-KABEL	(KEIN KOMMENTAR)

Einige Symbole haben eine vordefinierte Bedeutung, die durch den Benutzer nicht verändert werden sollte (System-Konstanten). Die beiden wichtigsten Symbole dieses Typs sind:

- **T** bezeichnet den booleschen Wert *True*;
- **NIL** bezeichnet den booleschen Wert *False* (und gleichzeitig auch die leere Liste).

Die Evaluation eines Symbols wird durch den Kontext gesteuert, in dem es vorkommt. Zunächst gilt:

Regel 2

Wenn keine andere Regel anwendbar ist, dann evaluiert ein Symbol zu seiner Wertbindung.

Beispiel (1-9)
Von den in Beispiel (1-8) beschriebenen Wertbindungen ausgehend, ergeben sich folgende Resultate:

PI $\Longrightarrow$ 3.14159
Name $\Longrightarrow$ "Otto Mustermann"

[9]Entsprechend der Struktur von Symbolen gibt es in Common-Lisp die Selektorfunktionen ***symbol-value***, ***symbol-function*** und ***symbol-plist***, die ein Symbol als Argument nehmen und als Wert die Wertbindung, Funktionsbindung, bzw. die Property–Liste dieses Symbols liefern.

A $\Longrightarrow$ **Fehlermeldung**
Die Evaluatierung von Symbolen, denen zuvor kein Wert zugewiesen wurde, führt zu einer entsprechenden Fehlermeldung und der Aktivierung eines **break-levels**.

Wertzuweisung. Es gibt in Common LISP eine ganze Reihe von Funktionen bzw. *special forms*[10] die es ermöglichen, einem Symbol einen Wert zuzuweisen. In den meisten Fällen wird die **SETQ** *special form* verwendet[11]:

SETQ	{*Symbol NeuerWert*}*	[Special Form]
SETQ nimmt eine Folge von *Symbol–NeuerWert* Paaren als Argumente und weist *Symbol* den Wert von *NeuerWert* zu (*value-binding*). *Symbol* wird nicht evaluiert. Der Wert der letzten *NeuerWert*–Form wird als Wert der SETQ–Form zurückgegeben.		

Beispiel (1-10)
(SETQ A 8) $\Longrightarrow$ 8
*Per Seiteneffekt wird das Symbol A an den Wert 8 gebunden (***value-binding***).*
A $\Longrightarrow$ 8

(SETQ A "Karl" B PI) $\Longrightarrow$ 3.14159
Per Seiteneffekt wird A an den Wert "Karl" und B an 3.14159 gebunden.
A $\Longrightarrow$ "Karl"
B $\Longrightarrow$ 3.14159

(SETQ A 8 B ($*$ A 3)) $\Longrightarrow$ 24
Sequentielle Evaluierung: Zunächst wird A an den Wert 8 gebunden. Diese Wertbindung wird dann verwendet, um den Wert von B zu berechnen.
A $\Longrightarrow$ 8
B $\Longrightarrow$ 24

Während SETQ nur die Veränderung der Wertbindung von Symbolen erlaubt, ermöglicht der SETF-Makro auch die Veränderung der Funktionsbindung und der *property-list* eines Symbols, sowie den Zugriff auf die Komponenten von Arrays und Listen.

[10] Der Unterschied zwischen Funktionen und *special forms* liegt u.a. in der Behandlung der Argumentbezeichner: Bei Funktionsaufrufen werden die Argumentbezeichner immer evaluiert; bei *special forms* muß das, wie man an QUOTE sieht, nicht der Fall sein. Würde QUOTE den Argumentbezeichner zunächst evaluieren, würde das in den meisten Fällen zu der Fehlermeldung führen, die man durch Verwendung von QUOTE gerade vermeiden will.

[11] "$*$" bezeichnet den Kleenschen Sternoperator; d.h. auf SETQ können beliebig viele Paare der Form $Symbol_1$ $NeuerWert_1$ ... $Symbol_n$ $NeuerWert_n$ folgen. Die geschweiften Klammern fungieren hier als metasprachliche Zeichen.

1.6.3 Listen

Wenn LISP eine Liste evaluiert, dann werden sequentiell alle Elemente der Liste evaluiert. Zu beachten ist allerdings, daß das erste Element der Liste als Funktionsname und die übrigen Elemente als Bezeichner der Argumente dieser Funktion aufgefaßt werden. Eine Liste repräsentiert in diesem Fall also einen Funktionsaufruf.

Nach Auswertung der Argumentsbezeichner wird die Funktion auf die Argumente angewendet und der aus der Funktionsanwendung resultierende Wert wird als Wert der Evaluation der Liste zurückgegeben.

Regel 3

Wenn ein Symbol als erstes Element einer zu evaluierenden Liste vorkommt, wird bei der Evaluation der Liste die *Funktionsbindung* des Symbols verwendet. Die dort gespeicherte Funktion wird auf das Resultat der Evaluierung aller übrigen Ausdrücke der Liste angewendet.

Beispiel (1-11)
(+ 9 6) $\Longrightarrow$ 15
*Zunächst wird die durch "+" bezeichnete Funktion lokalisiert (*Funktionsbindung*). Dann werden "9" und "6" evaluiert und anschließend die Funktion auf die errechneten Werte angewendet.*

(+ (− 5 7) 3) $\Longrightarrow$ 1
Zunächst wird die durch "+" bezeichnete Funktion lokalisiert. Das zweite Element der Liste ist selbst wieder eine Liste, also wird zunächst die durch "−" bezeichnete Funktion lokalisiert und anschließend auf die Werte von "5" und "7" angewendet. Als Zwischenergebnis erhalten wir –2. Dann wird 3 evaluiert und mit −2 addiert.
Bei komplexen Listen wird, wie das letzte Beispiel zeigt, die Regel 3 rekursiv angewendet. Allerdings wird jedes Objekt nur einmal evaluiert: evaluiert z.B. ein Symbol S_1 zu einem Symbol S_2, wird S_2 selbst nicht wieder evaluiert.

(8 9 8) $\Longrightarrow$ Fehlermeldung, da "8" keine Funktion bezeichnet.

Wie die Struktur von Symbolen schon vermuten läßt, kann ein Symbol gleichzeitig als Variable und als Funktionsname verwendet werden.

1.6.4 Unterdrücken und Erzwingen einer Evaluierung

In vielen Fällen ist es sinnvoll, die Evaluierung von S-Expressions zu verhindern, da sie zu unerwünschten Resultaten führen würde (z.B. '(8 9 10)'). Zu diesem Zweck gibt es in LISP eine *special form* mit dem Namen QUOTE:

QUOTE	*Objekt*	[Special Form]
QUOTE verhindert die Evaluierung des LISP–Objekts *Objekt* und liefert als Wert *Objekt*.		

Da diese *special form* sehr häufig benötigt wird, gibt es für sie eine Kurzschreibweise (*read-macro*): das Hochkomma ('). Es gilt also:

(QUOTE *Object*) ≡ '*Object*[12].

Andererseits gibt es eine Funktion, die wir im Zusammenhang mit der kurzen Charakterisierung der Funktionsweise des LISP-Interpreters erwähnt hatten, die eine zusätzliche Evaluierung erzwingt:

EVAL	*Form*	[Funktion]
Form wird wie jedes Argument eines Funktionsaufrufs evaluiert. EVAL erzwingt eine weitere Evaluierung; d.h., eine Evaluierung des Wertes von *Form*.		

Beispiel (1-12)

A	$\Longrightarrow$	Fehlermeldung - ungebundenes Symbol
(QUOTE A)	$\Longrightarrow$	A
'A	$\Longrightarrow$	A
(EVAL 'A)	$\Longrightarrow$	Fehlermeldung - ungebundenes Symbol
(EVAL "A)	$\Longrightarrow$	A
(SETQ A 0)	$\Longrightarrow$	0
A	$\Longrightarrow$	0
(EVAL 'A)	$\Longrightarrow$	0
'(8 9 10)	$\Longrightarrow$	(8 9 10)

[12] "≡" ist ein metasprachliches Zeichen und zu lesen als "ist äquivalent mit".

Kapitel 2

Funktionen

Nachdem wir die wichtigsten fr Common LISP geltenden syntaktischen und semantischen Regeln kennengelernt haben, werden wir im Anschlu an zwei kurze Exkurse die wichtigsten vordefinierten Funktionen zur Listenverarbeitung beschreiben und auf die Mglichkeiten eingehen, die LISP dem Benutzer zur Definition eigener Funktionen bietet.

2.1 Funktionen, Special Forms und Makros

Die dritte der im 1. Kapitel formulierten semantischen Regeln legt fest, da bei der Evaluierung einer Liste (Funktionsaufruf) immer alle Elemente der Liste evaluiert werden (das erste Element zu der anzuwendenden Funktion, alle anderen zu den Argumenten der Funktion). Diese Darstellung bedarf einer kleinen Korrektur:

Betrachtet man die Beispiele genau, in denen QUOTE bzw. SETQ als erstes Element einer Liste vorkommen, erkennt man, da es sehr wohl Flle gibt, in denen nicht alle Argumentbezeichner ausgewertet werden. So wird QUOTE immer dann verwendet, wenn es notwendig ist, die Evaluierung des Argumentbezeichners zu verhindern, und in einer SETQ-Form wird zwar immer das zweite Element eines Argument-Wert Paares ausgewertet, nicht aber das erste.

Funktionen, die, wenn sie aufgerufen werden, nicht alle ihre Argumentbezeichner auswerten, werden als *special forms* bezeichnet. Auerdem unterscheiden sie sich in vielen Fllen durch ihre Syntax von anderen Formen. Es gibt in Common LISP etwa 30 *special forms*.

Auer einer Funktion oder einer *special form* kann das erste Element einer Liste auch einen *Makro* bezeichnen. Makros werden nach besonderen Regeln evaluiert: Zunchst wird der Makro expandiert, und anschlieend wird der aus der Expansion resultierende Ausdruck evaluiert; d.h. wir haben zu unterscheiden zwischen:

- Funktionen;
- *Special Forms* und
- Makros.

Bei der Evaluierung einer Liste wird zunchst berprft, ob das erste Element der Liste zu einer *Special Form* evaluiert; wenn nicht, ob es einen Makro bezeichnet, und ist auch das nicht der Fall, ob es sich um einen Funktionsbezeichner handelt. Ist keiner der Tests positiv, wird eine Fehlermeldung zurckgegeben. Der Benutzer kann eigene Funktionen und Makros, nicht aber *Special Forms* definieren.

2.2 Semantische Klassifikation von Funktionen

Common Lisp ist ein sehr leistungsfhiger LISP-Dialekt, der dem Benutzer eine groe Zahl vordefinierter Funktionen zur Verfgung stellt (z.Z. mehr als 800). Um bei dieser Vielzahl von Funktionen den berblick bewahren zu knnen, ist es sinnvoll, sie nach semantischen Kriterien zu klassifizieren.

Kriterien, die sich zur Klassifikation von Funktionen gut eignen, sind einerseits die Objekte, die den Definitions- bzw. Wertebereich einer Funktion bilden, wie andererseits die Art der Operationen, die die Funktion auf den Objekten ihres Definitionsbereichs ausfhrt.

Definitionsbereich. Wenn man Funktionen zu einer Klasse zusammenfasst, die auf demselben Objekttyp operieren, dann erhlt man u.a. die folgenden Klassen:

- Funktionen, die auf Zahlen operieren:
  - Integerfunktionen
  - Realfunktionen
  - ...
- Zeichenfunktionen
- Zeichenkettenfunktionen
- Listenfunktionen
- Arrayfunktionen
- ...

Wertebereich. Die wichtigste durch ihren gemeinsamen Wertebereich charakterisierte Klasse von Funktionen ist die Klasse der *booleschen Funktionen*, hufig auch als **Prdikate** bezeichnet. Funktionen dieses Typs liefern als Wert einen Wahrheitswert. In Lisp werden Wahrheitswerte durch "T" bzw. "NIL" reprsentiert. Eine formale Schwche von LISP liegt allerdings darin, da jeder andere Wert als NIL als mit dem booleschen Wert TRUE quivalent behandelt wird; d.h. es gibt Prdikate, die statt "T" andere LISP-Objekte als Wert liefern[1].

Die drei wichtigsten Klassen von Prdikaten bilden die Typprdikate, die logischen Funktionen und die Vergleichs- bzw. Identittsfunktionen. Zur Bezeichnung von Prdikaten werden in LISP gewhnlich Symbolnamen verwendet, die mit dem Zeichen "p" enden; z.B. ***numberp***, ***standard-char-p*** ... .

Operationen. Die Funktionen, die auf komplexen Objekten wie z.B. Strings, Arrays und CONSes operieren, lassen sich nicht nur nach den Objekten ihres Definitionsbereichs ordnen, sondern auch bzgl. der von ihnen auf diesen Objekten ausgefhrten Operationen. Unter diesem Gesichtspunkt ist es blich, die folgenden drei Funktionstypen zu unterscheiden:

1. *Konstruktoren*:
   Funktionen, die Objekte dieses Typs erzeugen (z.B. CONS zur Erzeugung von (Paar-)Listen, MAKE-ARRAY zur Erzeugung von Arrays, ...);

2. *Selektoren*:
   Funktionen, die Komponenten aus diesen Objekten extrahieren (z.B. CAR und CDR, die als Wert das erste oder zweite Element eines CONSes liefern) und

3. *Modifikatoren*:
   Funktionen, die bereits erzeugte Objekte verndern (z.B. kann durch SETQ die Wertbindung eines Symbols verndert werden).

Die Unterscheidung zwischen diesen drei Funktionstypen ist gerade fr die Programmentwicklung von nicht zu unterschtzender Bedeutung: Gilt es, nicht-triviale Aufgabenstellungen zu meistern, ist es sinnvoll, zunchst eine dem Problem angemessene Datenstruktur zu whlen bzw. zu definieren. Im nchsten Schritt werden dann *Konstruktoren, Selektoren* und *Modifikatoren* fr diese Datenstruktur definiert. Erst danach wird auf der Basis des zuvor formulierte Algorithmus ein lauffhiges Programm erstellt. Das Resultat sind (in der Regel) klar strukturierte und stark modularisierte Programme, die einfach zu verstehen und zu modifizieren sind.

Es sollte nicht berraschen, da die drei genannten Mglichkeiten zur Kategorisierung von Funktionen zu keiner vollstndigen oder eindeutigen Klassifikation fhrt; d.h. es gibt Funktionen, die keiner der hier eingefhrten Klassen zugeordnet werden knnen (EVAL, QUOTE,

[1] So z.B. die Funktion MEMBER, die im positiven Fall die Restliste als Wert liefert, deren erstes Element das gesuchte Objekt ist.

LENGTH, ...), und es gibt Funktionen, die mehreren Klassen zuzuordnen sind: CONS z.B. ist eine Listenfunktion und ein Konstruktor.

2.3 Funktionen zur Listenverarbeitung

2.3.1 Konstruktoren

Neben der Funktion CONS, die wir bereits in der letzten Sitzung kennengelernt haben, gibt es in LISP noch zwei weitere wichtige Funktionen zur Erzeugung von Listen: **LIST** und **APPEND**.

LIST	*Argumente**	[Funktion]
LIST akzeptiert eine beliebige Zahl von Argumenten und liefert als Wert eine Liste, die alle durch die Argumente bezeichneten Objekte als Top-level-Elemente enthlt.		

APPEND	*Listen**	[Funktion]
APPEND akzeptiert eine beliebige Anzahl von Listen als Argumente und liefert als Wert die Liste, die durch Verknpfung der Listen entsteht - die Top-level-Elemente der Argument-Listen bilden die Top-level-Elemente der Ergebnisliste.		

Beispiel (2-1)

(LIST 'X 'Y 'Z)	$\Longrightarrow$	(X Y Z)
(LIST 33 () '(V W))	$\Longrightarrow$	(33 () (V W))
(LIST (* 9 7) "alte Gauner")	$\Longrightarrow$	(63 "alte Gauner")
(APPEND '(A B) '(C))	$\Longrightarrow$	(A B C)
(APPEND () '(A B) () '(C) ())	$\Longrightarrow$	(A B C)
(APPEND '(A) '((B)) '(((C))))	$\Longrightarrow$	(A (B) ((C)))
(APPEND (LIST (- 7 11) 'A) '(B C))	$\Longrightarrow$	(-4 A B C)
(LIST (APPEND '(A) '(B) '(C)))	$\Longrightarrow$	((A B C))

Charakteristisch fr CONS und die in diesem Abschnitt vorgestellten Funktionen ist, da sie Referenzen auf bestehende CONS-Strukturen nicht verndern, sondern neue erzeugen. Funktionen dieses Typs werden als *nicht-destruktive* Funktionen bezeichnet.

Beispiel (2-2)

```
> (SETQ *liste1* '(A B) *liste2* '(C D))
  (C D)
> (CONS *liste1* *liste2*)
  ((A B) C D)
> *liste1*
  (A B)
> *liste2*
  (C D)
> (APPEND *liste1* *liste2*)
  (A B C D)
> *liste1*
  (A B)
> *liste2*
  (C D)
> (list *liste1* *liste2*)
  ((A B) (C D))
> *liste1*
  (A B)
> *liste2*
  (C D)
```

Wie diese Beispiele zeigen, hat die Verwendung der Konstruktoren CONS, APPEND, LIST keinen Einflu auf die Wertbindung von Symbolen.

2.3.2 Selektoren

An Selektoren fr Listen herrscht in Common LISP kein Mangel. Neben den 'traditionellen' Selektoren **CAR** und **CDR** gibt es zahlreiche Selektoren mit weniger rtselhaften Namen.

CAR	*Liste*	[Funktion]
Die Funktion CAR liefert das erste Element von *Liste*; d.h. das Objekt, auf das der CAR-Pointer der ersten CONS-Zelle von *Liste* weist.		

CDR	*Liste*	[Funktion]
Die Funktion CDR liefert eine Liste, die alle Elemente von *Liste* auer dem ersten enthlt; d.h. das Objekt, auf das der CDR-Pointer der ersten CONS-Zelle von *Liste* weist.		

Da die Selektorfunktionen CAR und CDR sehr hufig verwendet werden, gibt es die Mglichkeit, CAR/CDR-Sequenzen abzukrzen. So gilt:

$$\begin{array}{l} (\text{CAAR } \alpha) \equiv (\text{CAR } (\text{CAR } \alpha)) \\ (\text{CDDR } \alpha) \equiv (\text{CDR } (\text{CDR } \alpha)) \\ (\text{CADR } \alpha) \equiv (\text{CAR } (\text{CDR } \alpha)) \\ \quad \vdots \qquad \vdots \qquad \vdots \end{array}$$

Die Grenzen dieses kompositionellen Verfahrens sind implementationsspezifisch.

Neben CAR, CDR und C...R gibt es in Common LISP auerdem noch die Funktionen FIRST, SECOND, THIRD, ... und REST. Es gilt:

$$\begin{array}{ll} (\text{FIRST } \alpha) & \equiv (\text{CAR } \alpha) \\ (\text{SECOND } \alpha) & \equiv (\text{CADR } \alpha) \\ \quad \vdots & \quad \vdots \quad \vdots \\ (\text{REST } \alpha) & \equiv (\text{CDR } \alpha) \end{array}$$

Beispiel (2-3)

$$\begin{array}{rcl} (\text{CAR } ()) & \Longrightarrow & \text{NIL} \\ (\text{CAR } '(\text{A B C})) & \Longrightarrow & \text{A} \\ (\text{CAR } '((\text{A}) \text{ B } ((\text{C})))) & \Longrightarrow & (\text{A}) \\ (\text{CDR } ()) & \Longrightarrow & \text{NIL} \\ (\text{CDR } '(\text{A B C})) & \Longrightarrow & (\text{B C}) \\ (\text{CADDAR } '((\text{A B C}) (\text{B}) ())) & \Longrightarrow & \text{C} \\ (\text{THIRD } '(\text{A B C})) & \Longrightarrow & \text{C} \\ (\text{FOURTH } '(\text{A B C})) & \Longrightarrow & \text{NIL}^2 \end{array}$$

Eine andere Mglichkeit, auf best. Elemente bzw. Teile einer Liste zuzugreifen, liefern die Funktionen **NTH** und **NTHCDR**:

NTH	*N* *Liste*	[Funktion]
Diese Funktion liefert das *N*-te Element von *Liste*, wobei das *CAR* der Liste den Index 0 hat. Bei berschreitung der Listenlnge wird NIL zurckgegeben.		

NTHCDR	*N* *Liste*	[Funktion]
Diese Funktion fhrt *N*-mal die Operation CDR auf *Liste* aus und liefert die daraus resultierende Liste als Wert.		

[2] Grundstzlich knnen alle diese Funktionen nicht nur auf Listen, sondern auf alle CONSes, also auch Paarlisten, angewendet werden; so gilt z.B.:
(CAR '(A . B)) $\Longrightarrow$ A und
(CDR '(A . B)) $\Longrightarrow$ B.

Beispiel (2-4)

```
(NTH 0 '(A B C))    ⟹  A
 (NTH 2 '(A B C))   ⟹  C
(NTH 3 '(A B C))    ⟹  NIL
(NTHCDR 1 '(A B C)) ⟹  (B C)
 (NTHCDR 2 '(A B C)) ⟹  (C)
```

Als letzte Selektorfunktionen fr Listen betrachten wir noch **LAST** und **BUTLAST**:

LAST	*Liste*	[Funktion]
Diese Funktion liefert das letzte CONS von *Liste* als Wert; d.h. eine Liste, die nur das letzte Element von *Liste* enthlt.		

BUTLAST	*Liste* *&optional N*	[Funktion]
Diese Funktion liefert eine Liste, die alle Elemente auer dem letzten/den letzten N Elementen von *Liste* enthlt.		

Beispiel (2-5)

```
 (LAST '(A B C))    ⟹  (C)
(LAST '(A B . C))   ⟹  (B . C)
(LAST ())           ⟹  NIL
 (BUTLAST '(A B C))    ⟹  (A B)
(BUTLAST '(A (B C)))   ⟹  (A)
(BUTLAST '(A B C) 2)   ⟹  (A)
(BUTLAST '(A B C) 4)   ⟹  NIL
```

2.3.3 Modifikatoren

Whrend sich SETQ nur dazu verwenden lt, die Wertbindung eines Symbols zu verndern, kann durch das SETF-Makro nicht nur Wert- und Funktionsbindung von Symbolen, sondern auch die Struktur von Listen modifiziert werden: Als erstes Argument von SETF ist nicht nur ein Symbol zulssig, sondern auch ein Funktionsaufruf, dessen erstes Element ein eine Selektorfunktion bezeichnender Symbolname wie CAR, CDR,CAAR, ... bzw. FIRST, SECOND, ... ist[3]. Solche als erstes Argument von SETF zulssigen Formen, die den Zugriff auf spezifische Komponenten von komplexen Objekten erlauben, werden als *generalisierte Variablen* bezeichnet. Das durch den Funktionsaufruf selektierte Listenelement wird in diesem Fall durch das Objekt ersetzt, zu dem der zweite Argumentbezeichner der SETF-Form evaluiert. Im Gegensatz zu den anderen in dieser Sitzung behandelten Funktionen, ist SETF (ebenso wie SETQ) *destruktiv*; d.h. es verndert bestehende Objekte bzw. Referenzbeziehungen (vgl. Beispiel (2-6)).

Bislang haben wir SETQ/SETF nicht nur zur *nderung* von Objekten verwendet (als Modifikator), sondern auch, um Variablen zu *erzeugen* (als Konstruktor). Diese Verwendung von SETQ/SETF gilt in neueren LISP-Dialekten, in denen es spezielle Anweisungen zum Deklarieren von Variablen und Parametern gibt, als schlechter Programmierstil. Wir werden aus diesem Grund nach Einfhrung dieser Anweisungen im nchsten Kapitel beide Funktionen ausschlielich als Modifikatoren verwenden.

Beispiel (2-6)

```
> (SETF *Obst* '(apfel birne kirsche))
  (APFEL BIRNE KIRSCHE)
> (SETF *Obst* (list 'pfirsich 'banane))
  (PFIRSICH BANANE)
> *Obst*
  (PFIRSICH BANANE)
> (SETF *Gepaeck* '(koffer aktentasche plastiktuete))
  (KOFFER AKTENTASCHE PLASTIKTUETE)
> (SETF (SECOND *Gepaeck*) 'regenschirm)
  REGENSCHIRM
> *Gepaeck*
```

[3]Die mit SETF gegebenen Mglichkeiten sind damit bei weitem noch nicht ausgeschpft, s. Steele[1990], S.123-29. Die Mchtigkeit der Funktion SETF macht eine Reihe von Funktionen berflssig,die in anderen LISP-Dialekten zur Modifikation von CONS-Strukturen verwendet werden; so z.B. die Funktionen RPLACA und RPLACD, die die Vernderung des CARs bzw. CDRs einer CONS-Struktur erlauben.

(KOFFER REGENSCHIRM PLASTIKTUETE)

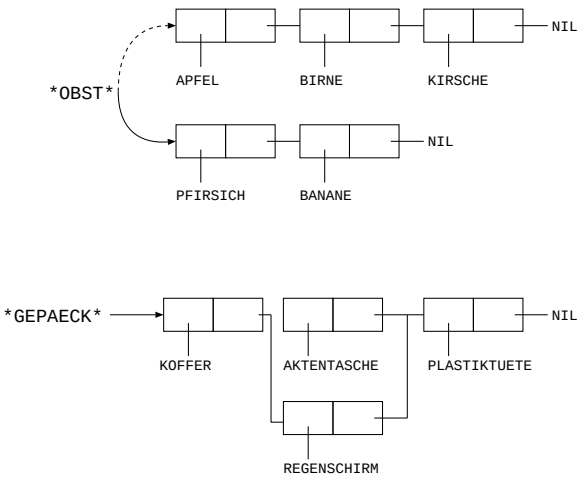


Abbildung (2-1)

2.3.4 Weitere Listenfunktionen

Nicht alle der gngigen Listenfunktionen lassen sich in das Schema *Konstruktor-Selektor-Modifikator* einfgen. Dieses gilt z.B. fr die Funktionen **REVERSE**[4], **LENGTH** und **MEMBER**:

REVERSE	*Liste*	[Funktion]
Diese Funktion liefert eine Liste, die alle Elemente von *Liste* in umgekehrter Reihenfolge enthlt.		

LENGTH	*Liste*	[Funktion]
Diese Funktion liefert als Wert die Anzahl der Elemente von *Liste*.		

Beispiel (2-7)

(REVERSE '(A B C)) $\Longrightarrow$ (C B A)
(REVERSE '(A (B C)) $\Longrightarrow$ ((B C) A)
(LENGTH '(A B C)) $\Longrightarrow$ 3
(LENGTH '(A (B C)) $\Longrightarrow$ 2

[4]REVERSE knnte auch als Listenkonstruktor behandelt werden, da diese Funktion eine Kopie der ursprnglichen Liste erzeugt, in der die Elemente in umgekehrter Reihenfolge vorkommen.

Das Prdikat MEMBER prft, ob ein Objekt in einer gegebenen Liste enthalten ist:

MEMBER *Objekt Liste* [Funktion]
Diese Funktion sucht nach dem ersten Vorkommen von *Objekt* in *Liste*. Kommt es in *Liste* nicht vor, liefert MEMBER NIL; sonst die Restliste von *Liste*, die mit dem ersten Vorkommen von *Objekt* beginnt. Die Suche beschrnkt sich auf das Top-Level und beschrnkt sich zunchst nur auf einfache, unstrukturierte Objekte.

Beispiel (2-8)

```
(MEMBER 'B '(A B C))          ⟹  (B C)
(MEMBER 'A '(A B C))          ⟹  (A B C)
(MEMBER 'D '(A B C))          ⟹  NIL
(MEMBER 'B '(A (B) C))        ⟹  NIL
(MEMBER '(B) '((A) (B) (C)))  ⟹  NIL
```

2.4 Benutzerdefinierte Funktionen

Wir erinnern uns: Ein LISP-Programm P besteht aus einer Folge von Funktionen. Um P auszufhren, wird die Hauptfunktion von P aufgerufen. Dieser Funktionsaufruf hat fhrt dazu, da weitere in P definierte Funktionen aufgerufen werden, die ihrerseits wieder weitere Funktionen aufrufen knnen (...). Die Entwicklung eines LISP-Programms besteht also in der Definition von Funktionen, die eine Lsung der zugrundeliegenden Aufgabenstellung erlauben.

2.4.1 Abstraktion

Der der Definition von Funktionen und Prozeduren zugrundeliegende Proze kann als ein **Abstraktionsproze** aufgefasst werden[5]

[5]Vgl. Michaelson[1988], S.15-20 und Stoyan&Grz[1984], S.39/40., dessen Ausgangspunkt konkrete Berechnungsvorschriften bilden, aus denen dann durch *Generalisierung* (Ersetzung von Konstanten durch Variablen) Prozeduren gewonnen werden.

Beispiel (2-9)
Um z.B. die Flche eines Kreises mit einem Radius von 2,7 cm zu berechnen, mu die Berechnungsvorschrift $2{,}7^2$ * 3.14159 ausgewertet werden. Eine Prozedur[6] unterscheidet sich von einer konkreten Berechnungsvorschrift darin, da der Anwendungsbereich der sie konstituierenden Operationssequenz nicht ein bestimmtes einzelnes Objekt ist, sondern die Klasse aller Objekte, die bzgl. der Operationen in dem gegebenen Problemzusammenhang quivalent sind (quivalenzklassenbildung): hier z.B. alle positiven reellen Zahlen.

Dieser konzeptuelle Unterschied findet seinen Niederschlag in der Ersetzung der in der Berechnungsvorschrift verwendeten Konstante durch eine Variable wie z.B. *radius*:

$$\text{radius}^2 * 3.14159$$

Ein zweiter Schritt, der bei der Definition einer Prozedur in den meisten Programmiersprachen erforderlich ist (nicht aber in LISP), besteht darin, der Prozedur einen Namen zuzuordnen:

$$\textbf{Kreisflche}(\text{radius}) :=_{\text{def}} \text{radius}^2 * 3.14159$$

2.4.2 Namenlose Funktionen

In LISP ist es mglich, Funktionen zu definieren, ohne ihnen einen Namen zu geben. Diese Funktionen werden als *anonyme Funktionen* bezeichnet und durch Verwendung eines Lambda-Ausdrucks notiert. Innerhalb des Lambda-Kalkls knnte unsere Funktion **Kreisflche** unter Vernachlssigung von Details durch folgenden Ausdruck notiert werden:

$$(\lambda\text{radius } (\text{radius}^2 * 3.14159))$$

Lambda-Ausdrcke, die in LISP zur Bezeichnung anonymer Funktionen verwendet werden, bestehen aus drei Komponenten: dem Symbol *lambda*, einer Liste von Variablen, der sogen. *Lambda-Liste*, die Namen fr die Parameter der Funktion spezifiziert, und einer Folge von Formen[7]:

lambda	(*variable**)	{*form*}*	[Lambda-Form]
Wenn die durch den Lambda-Ausdruck bezeichnete Funktion ausgefhrt wird, werden - nach Bindung der Parameter an die beim Funktionsaufruf verwendeten Argumente - die Formen sequentiell evaluiert und der Wert der letzten Form als Funktionswert zurckgegeben.			

[6] Wir vernachlssigen in diesem Zusammenhang parameterlose Prozeduren, die einfach eine Folge konkreter Berechnungsvorschriften zusammenfassen.

[7] Ein Lambda-Ausdruck ist keine Form; d.h. er kann nicht sinnvoll evaluiert werden. Er bezeichnet eine Funktion.

Die Funktion Kreisflche lt sich notieren als:

(lambda (radius) (* radius radius 3.14159))

Als Funktionsaufruf erhalten wir z.B.:

((lambda (radius) (* radius radius 3.14159)) 2.7) $\Longrightarrow$ 22.90220

2.4.3 Funktionen mit Namen

Natrlich ist es sehr viel praktischer, Funktionen durch ein Symbol bezeichnen zu knnen und so nicht bei jedem Funktionsaufruf einen unter Umstnden komplexen Lambda-Ausdruck verwenden zu mssen. Wie kann einem Symbol eine Funktionsdefinition so zugeordnet werden, da sie die *Funktionsbindung* des Symbols bildet und damit das Symbol in Funktionsaufrufen als Funktionsbezeichner verwendet werden kann? Es ist naheliegend vorzuschlagen, mit SETQ bzw. SETF einem Symbol einen Lambda-Ausdruck als Wert zuzuordnen:

(setq kreisflaeche (lambda (radius) (* radius radius 3.14159)))

Das Resultat lt allerdings zu wnschen brig; denn die Evaluierung der Form(kreisflaeche 2.7) fhrt zu einer Fehlermeldung wie "*undefined function kreisflaeche*". Der mit der Evaluierung der SETQ-Form verbundene Seiteneffekt bestand in einer Vernderung der *Wertbindung*, nicht aber der *Funktionsbindung* des Symbols KREISFLAECHE:

```
> kreisflaeche
  (LAMBDA (RADIUS) (* RADIUS RADIUS 3.14159))
```

Eine Alternative bildet die Verwendung der Funktion *symbol-function*, die den Zugriff auf die *Funktionsbindung* eines Symbols ermglicht:

```
(setf (symbol-function 'kreisflaeche)
  (lambda (radius) (* radius radius 3.14159)))
```

Noch einfacher ist allerdings die Verwendung des **DEFUN**-Makros (**de***fine* **fun***ction*), wenn auch das Resultat in beiden Fllen dasselbe ist:

DEFUN *Name* *Lambda-Liste* {*Form*}* [Makro]
Die Evaluierung einer *Defun*-Form hat zur Folge, da das Symbol *Name* zum globalen Namen fr die Funktion wird, die durch den Lambda-Ausdruck (lambda *Lambda-Liste* {*Form*}*) bezeichnet wird.

(defun *name lambda-liste* {*form*}*)
$\equiv$
(setf (symbol-function *name*) (lambda *lambda-liste* {*form*}*).

Beispiel (2-10)

```
(defun kreisflaeche (radius)          ; Funktionskopf
  (* radius radius 3.14159))          ; Funktionskrper
```

In einer Funktionsdefinition unterscheidet man zwischen dem *Funktionskopf*, der Funktionsname und Parameterliste umfat, und dem *Funktionskrper*, d.h. der Sequenz von Anweisungen (auch *Anweisungsblock* genannt), die bei jedem Aufruf der Funktion auszufhren sind.

2.4.4 Bindung und Umgebung

Eine genauere Analyse der Evaluierung von Funktionsaufrufen macht es erforderlich, einige neue Begriffe einzufhren:

- Mit **Bindung** (*binding*) bezeichnet man in LISP die Beziehung zwischen einer Variablen und dem ihr zugeordneten Wert.
- Eine **Umgebung** (*environment*) besteht aus der Menge aller zu einem gegebenen Zeitpunkt bestehenden *Bindungen*.

Die in der Definition der Funktion in der Lambda-Liste verwendeten Variablen werden als **formale Parameter**, die bei dem Funktionsaufruf verwendeten Argumente als **aktuelle Parameter** bezeichnet.

Wie vollzieht sich die Evaluierung eines Funktionsaufrufs wie z.B. (kreisflaeche 2.7)?

1. Zunchst werden, wie wir wissen, die Ausdrcke von links nach rechts ausgewertet: *kreisflaeche* evaluiert zu dem durch dieses Symbol bezeichneten Funktionsobjekt; 2.7 evaluiert zu 2.7.
2. Anschlieend wird der formale Parameter *radius* an den aktuellen Parameter 2.7 gebunden. Durch diese Bindung wird eine neue Umgebung erzeugt. Die alte Umgebung bleibt zwar erhalten, ist aber whrend der Ausfhrung des Funktionsaufrufs nicht aktiv, sondern wird es erst wieder nach seiner Ausfhrung. Die neue Umgebung besteht aus den Parameterbindungen und allen Bindungen der alten Umgebung, die nicht durch die Parameterbindungen *verschattet* werden.
3. In dieser neuen Umgebung wird der den Funktionskrper bildende Anweisungsblock ausgewertet und der aus dieser Auswertung resultierende Wert als Wert des Funktionsaufrufs zurckgegeben.

4. Abschlieend wird wieder die alte Umgebung aktiviert[8]

Die Ausfhrung eines Funktionsaufrufs lt sich also als eine Sequenz auffassen, die aus folgenden vier Schritten besteht:

1. Aufbau einer neuen Umgebung;
2. Aktualisierung der neuen Umgebung;
3. Auswertung des Funktionskrpers und
4. Aktualisierung der alten Umgebung.

2.4.5 Dokumentation

Nur wer sich vor Freizeit nicht zu retten wei oder seit Jahren mit Leidenschaft Hieroglyphen entschlsselt, kann sich ber nicht oder schlecht dokumentierte Funktionen und Programme freuen. Dieser Abschnitt formuliert einige Mindestanforderungen, die gut dokumentierte Programme erfllen sollten[9]:

1. Einer Funktionsdefinition sollten (mindestens) folgende Angaben vorangestellt werden:
   - Aufgabe der Funktion;
   - die Parameter der Funktion (mit Namen & Wertebereich);
   - die Werte, die die Funktion zurckliefert und
   - die Seiteneffekte, die durch die Ausfhrung der Funktion erzielt werden.
2. In komplexen Funktionen kann es auerdem sinnvoll sein, auch einzelne Anweisungen innerhalb der Funktion zu kommentieren. Dabei sollte die *Semikolon-Konvention* beachtet werden.
3. Hilfreich ist es auch, durch Einrcken von Anweisungen die Struktur von Funktionsdefinitionen visuell zu unterstreichen.
4. Auerdem empfiehlt es sich, bei Funktionen und ihren Parametern selbstdokumentierende Namen zu verwenden.
5. Die in Common LISP blichen Notationskonventionen sollten respektiert werden: So sollten Namen globaler Variablen in "*" eingeschlossen werden, Prdikatsnamen mit "p" "-p" enden. Die Beachtung dieser Konventionen erhht die Lesbarkeit von Programmen erheblich (zumindest fr diejenigen, die mit ihnen vertraut sind).

[8] Die Evaluierung von Formen, die Seiteneffekte verursachen, kann indirekt zu einer Vernderung der alten Umgebung fhren; z.B. wenn durch SETQ bzw. SETF Parameterbindungen modifiziert werden.

[9] Da wir uns in den folgenden Sitzungen zunchst nur mit der Entwicklung einzelner Funktionen beschftigen werden, ist es noch verfrht, ber Konzepte zur Programmdokumentation nachzudenken.

Beispiel (2-11)

```
;;; AUFGABE:     Berechnung von Kreisflchen
;;; PARAMETER: radius - reelle Zahl
;;; WERT:        reelle Zahl[10]
(defun kreisflaeche (radius)
  (* radius radius 3.14159))
```

Die *Semikolon-Konvention*

```
;   Kommentar fr den in der gleichen Zeile stehenden Kode.
;;  Kommentar fr den Kode der folgenden Zeilen.
;;; Kommentar, der sich auf die folgende Funktionsdefinition bezieht.
;;;; Kommentar am Anfang der Datei, der sich auf das gesamte
;;;; nachfolgende Programm bezieht.
```

Beispiel (2-12)

```
;;; Zum Schlu eine ganz tolle Funktion:
(defun t1 (x); eine Funktion mit einem besonders vielsagenden Namen
  ;; jetzt wird gerechnet:
  (* x x 3.1456))
```

[10] Es ist ausreichend, das Auftreten von Seiteneffekten zu dokumentieren; d.h. in unserem Beispiel sind keine Angaben erforderlich.

Kapitel 3

Prädikate und Kontrollstrukturen

Nachdem in den vergangenen beiden Kapiteln vor allem die konzeptionellen Grundlagen von Common LISP im Vordergrund standen, werden wir im folgenden eine Reihe von Funktionen einführen, die für die Entwicklung nicht-trivialer Programme unentbehrlich sind. Systematisch betrachtet, können sie in zwei Klassen zusammengefaßt werden:

1. Prädikate (Typ-, Identitäts- und logische Prädikate) und
2. Funktionen, durch die in LISP die Kontrollstrukturen realisiert werden, die für prozedurale Programmiersprachen charakteristisch sind.

3.1 Prädikate

Prädikate sind strenggenommen Funktionen, deren Wertebereich ausschließlich aus den beiden booleschen Werten *true* und *false* besteht. Da in LISP jedes Objekt außer NIL dazu verwendet werden kann, den booleschen Wert *true* zu repräsentieren, gibt es einige LISP-Prädikate (z.B. logische Prädikate wie OR und AND), die, wenn die durch sie formulierte Bedingung erfüllt ist, nicht *true*, sondern ein geeignetes LISP-Objekt als Wert liefern.

3.1.1 Typprädikate

In Kapitel 1 haben wir die wichtigsten Klassen von Objekten eingeführt, die in Common LISP unterschieden werden. Typprädikate erlauben es festzustellen, ob ein Objekt O Element einer Objektklasse K ist oder nicht. Zu den wichtigsten Typprädikaten gehören: NUMBERP, SYMBOLP, ATOM, LISTP und ENDP[1].

NUMBERP	*Objekt*	[Funktion]
NUMBERP evaluiert zu T, gdw. *Objekt* eine Zahl beliebigen Typs ist; sonst zu NIL.		

Für die verschiedenen in Common LISP verfügbaren Zahlentypen (natürliche Zahlen, reelle Zahlen, ...) gibt es weitere Typprädikate (INTEGERP, REALP, ...). Um festzustellen, ob ein Objekt ein Symbol ist, kann man das SYMBOLP-Prädikat verwenden:

SYMBOLP	*Objekt*	[Funktion]
SYMBOLP liefert T, gdw. *Objekt* ein Symbol ist; sonst NIL.		

Zahlen, Zeichen, Zeichenketten, Symbole und die leere Liste werden als *atomare* Objekte bezeichnet.

ATOM	*Objekt*	[Funktion]
ATOM liefert T, gdw. *Objekt* kein CONS ist; sonst NIL.		

Alle Listen, mit Ausnahme der leeren Liste, sind *nicht-atomare* Objekte. Die leere Liste ist gleichzeitig eine Liste und ein atomares Objekt, da sie kein CONS ist. Dieser Fall zeigt, daß die Klassifikation der LISP-Objekte nicht eindeutig ist.

LISTP	*Objekt*	[Funktion]
LISTP liefert T, gdw. *Objekt* ein CONS oder die leere Liste ist; sonst NIL. Es wird nicht geprüft, ob es sich bei *Objekt* um eine Liste oder ein *dotted pair* handelt.		

Es gibt verschiedene Möglichkeiten, um zu überprüfen, ob ein Objekt die leere Liste repräsentiert. Normalerweise sollte zu diesem Zweck das Prädikat ENDP verwendet werden:

ENDP	*Objekt*	[Funktion]
ENDP evaluiert zu T, gdw. *Objekt* die leere Liste ist und zu NIL wenn *Objekt* ein CONS ist. In allen anderen Fällen erhält man eine Fehlermeldung.		

[1] Prädikatsnamen enden, wie die Namen dieser Systemprädikate zeigen, typischerweise mit "p" (bzw. "-p").

Neben ENDP gibt es auch noch das Prädikat NULL. Im Unterschied zu ENDP evaluiert eine NULL-Form *immer* zu NIL, wenn das Argument von NULL nicht zu NIL evaluiert; d.h. anders als bei ENDP führen atomare Argumente nicht zu einer Fehlermeldung.

Beispiel (3-1)

(numberp 'pi)	$\Longrightarrow$	NIL
(numberp pi)	$\Longrightarrow$	T
(numberp 9)	$\Longrightarrow$	T
(numberp -7.111)	$\Longrightarrow$	T
(numberp #\z)	$\Longrightarrow$	NIL
(symbolp 'pi)	$\Longrightarrow$	T
(symbolp pi)	$\Longrightarrow$	NIL
(symbolp #\z)	$\Longrightarrow$	NIL
(symbolp "Murx")	$\Longrightarrow$	NIL
(symbolp 'Ein-toller_Symbol.Name)	$\Longrightarrow$	T
(atom 'pi)	$\Longrightarrow$	T
(atom pi)	$\Longrightarrow$	T
(atom ())	$\Longrightarrow$	T
(atom #\z)	$\Longrightarrow$	T
(atom "Murx")	$\Longrightarrow$	T
(atom '(a b))	$\Longrightarrow$	NIL
(listp ())	$\Longrightarrow$	T
(listp '(a . b))	$\Longrightarrow$	T
(listp '(a b))	$\Longrightarrow$	T
(listp 9)	$\Longrightarrow$	NIL
(endp ())	$\Longrightarrow$	T
(endp '(a . b))	$\Longrightarrow$	NIL
(endp '(a b))	$\Longrightarrow$	NIL
(endp 9)	$\Longrightarrow$	*Fehlermeldung*

3.1.2 Identitätsprädikate

Zunächst mag die Unterscheidung verschiedener Identitätsprädikate wenig plausibel erscheinen: Entweder zwei Objekte sind identisch oder sie sind es nicht. Die Unterschiede zwischen den Prädikaten EQ, EQL und EQUAL ergeben sich durch die Kriterien, die jeweils verwendet werden, um zu entscheiden, ob zwei Objekte identisch sind oder nicht:

- Eine EQ-Form evaluiert nur dann zu T, wenn beide Argumentbezeichner dasselbe Symbol bezeichnen.
- Eine EQL-Form liefert auch dann T, wenn sie dasselbe atomar Objekte bezeichnen (Zahlen, Zeichen).

- Eine EQUAL-Form schließlich evaluiert auch dann zu T, wenn sie komplexe, aber gleichstrukturierte Objekte (Listen, Arrays, ...) bezeichnen.

Neben diesen drei Identitätsprädikaten, die für beliebige LISP-Objekte verwendet werden können, gibt es für jeden Klasse von atomaren Objekten spezielle Vergleichs- und Identitätsprädikate:

(a) für Zahlen:
= > < <= >= /=

(b) für Zeichen:
CHAR= CHAR-EQUAL CHAR> CHAR>= CHAR< CHAR<= CHAR/=

EQ	*Objekt*$_1$ *Objekt*$_2$	[Funktion]
EQ evaluiert zu T, gdw. es sich bei *Objekt*$_1$ und *Objekt*$_2$ um dasselbe Objekt handelt; sonst zu NIL.		

EQL	*Objekt*$_1$ *Objekt*$_2$	[Funktion]
EQL liefert T, gdw. *Objekt*$_1$ mit *Objekt*$_2$ identisch ist (EQ) oder es sich um Zahlen desselben Typs mit gleichem Wert handelt oder um Zeichenobjekte, die dasselbe Zeichen repräsentieren.		

EQUAL	*Objekt*$_1$ *Objekt*$_2$	[Funktion]
EQUAL liefert T, gdw. *Objekt*$_1$ und *Objekt*$_2$ gleichartige atomare Objekte sind (EQL) oder es sich bei ihnen um komplexe Objekte handelt, die strukturell isomorph sind.		

Beispiel (3-2)

(eq 3 3.0)	$\Longrightarrow$	NIL
(eq (* 3 3) 9)	$\Longrightarrow$	???[2]
(eq #\z #\z)	$\Longrightarrow$	???
(eq "Murx" "Murx")	$\Longrightarrow$	???
(eq 'a 'a)	$\Longrightarrow$	T
(eq 'a 'b)	$\Longrightarrow$	NIL
(eq (cons 'a 'b) (cons 'a 'b))	$\Longrightarrow$	NIL
(eq '(a . b) '(a . b))	$\Longrightarrow$	???
(eq '(a b) '(a b))	$\Longrightarrow$	???
(eql 3 3.0)	$\Longrightarrow$	NIL
(eql (* 3 3) 9)	$\Longrightarrow$	T

(eql #\z #\z)	$\Longrightarrow$	T
(eql "Murx" "Murx")	$\Longrightarrow$	???
(eql 'a 'a)	$\Longrightarrow$	T
(eql 'a 'b)	$\Longrightarrow$	NIL
(eql (cons 'a 'b) (cons 'a 'b))	$\Longrightarrow$	NIL
(eql '(a . b) '(a . b))	$\Longrightarrow$	???
(eql '(a b) '(a b))	$\Longrightarrow$	???
(equal 3 3.0)	$\Longrightarrow$	NIL
(equal (* 3 3) 9)	$\Longrightarrow$	T
(equal #\z #\z)	$\Longrightarrow$	T
(equal "Murx" "Murx")	$\Longrightarrow$	T
(equal 'a 'a)	$\Longrightarrow$	T
(equal 'a 'b)	$\Longrightarrow$	NIL
(equal (cons 'a 'b) (cons 'a 'b))	$\Longrightarrow$	T
(equal '(a . b) '(a . b))	$\Longrightarrow$	T
(equal '(a b) '(a b))	$\Longrightarrow$	T

3.1.3 Logische Prädikate

Logische Prädikate ermöglichen es, aus einfachen Bedingungen beliebig komplexe Bedingungen zu bilden. In LISP stehen dazu die drei Funktionen NOT, AND und OR zur Verfügung, die weitgehend den der aus der Logik bekannten Operationen der Negation, Konjunktion und Disjunktion entsprechen. Durch geschickte Kombination logischer Prädikate lassen sich die meisten der Kontrollstrukturen simulieren, die wir im nächsten Abschnitt einführen werden:

NOT	*Objekt*	[Funktion]
NOT evaluiert zu T, gdw. *Objekt* = NIL ist; sonst zu NIL.		

NOT ist äquivalent mit NULL, sollte aber immer dann verwendet werden, wenn es darum geht, logische Verknüpfungen und Tests zu formulieren. Anders als NOT sind AND und OR keine einstellige Funktionen: Sie akkzeptieren eine beliebige Zahl von Argumenten.

AND	$\{Form\}^*$	[Funktion]
AND evaluiert die Formen nacheinander von links nach rechts. Sobald eine Form zu NIL evaluiert, wird die Evaluation der AND-Form abgebrochen und NIL als Wert zurückgeliefert; sonst evaluiert sie zum Wert der letzten Form.		

[2]Durch den Common LISP-Standard nicht festgelegt; d.h. das Ergebnis kann hängt von der jeweiligen Implementierung ab.

OR	$\{Form\}^*$	[Funktion]
OR evaluiert die Formen nacheinander von links nach rechts. Sobald eine Form zu einem anderen Wert als NIL evaluiert, wird die Evaluation der OR-Form abgebrochen und der Wert dieser Form zurückgeliefert; sonst wird der Wert der letzten Form zurückgeliefert.		

Beispiel (3-3)

(not ())	$\Longrightarrow$	T
(not '(a b))	$\Longrightarrow$	NIL
(not "Pleite")	$\Longrightarrow$	NIL
(not 99)	$\Longrightarrow$	NIL
(and 1 3 5 7)	$\Longrightarrow$	7
(and 99 "alte" () vw)	$\Longrightarrow$	NIL
(and 99 "alte" vw ())	$\Longrightarrow$	"error: ..."
(and 99 "alte" (not ()) 'vw)	$\Longrightarrow$	VW
(or 1 3 5 7)	$\Longrightarrow$	1
(or (= 1 2) (> 2 7))	$\Longrightarrow$	NIL
(or (and (<= 2 3) (not (>= 2 3))) (* 2 3))	$\Longrightarrow$	T
(and (or (< 2 3) (>= 2 3)) (* 2 3))	$\Longrightarrow$	6

3.2 Kontrollstrukturen

Es ist nicht ganz unproblematisch Kontrollstrukturen, wie man sie aus prozeduralen Programmiersprachen wie z.B. PASCAL, MODULA-2 oder C kennt, durch Funktionen zu realisieren: Charakteristisch für die Ausführung eines Funktionsaufrufs in LISP ist, daß *alle* Argumentbezeichner evaluiert werden. Der Aufruf etwa einer der IF-Anweisung korrespondierenden Funktion führte so in jedem Fall zur Evaluierung der der *Then*–Anweisung und der *Else*–Anweisung entsprechenden Argumentbezeichner. Vernachlässigt man für den Augenblick Effizienzüberlegungen, so bleibt die Möglichkeit von für die Programmentwicklung unangenehmen Seiteneffekten. Aus diesem Grund werden in LISP alle Kontrollstrukturen, die wir in diesem Abschnitt einführen werden, als *Special Forms* bzw. Makros realisiert.

Als elementare Kontrollstrukturen läßt sich die Zusammenfassung einer Folge von Anweisungen zu einer (komplexen) Anweisung auffassen. Diese Kontrollstruktur wird in LISP durch PROGN repräsentiert:

PROGN	$\{Form\}^*$	[Special Form]
PROGN nimmt eine Folge von Formen als Argumente, evaluiert sie nacheinander und liefert als Wert den Wert der letzten Form.		

Bei der Definition einer Funktion werden die Anweisungen des Funktionskörpers durch ein implizites PROGN miteinander verknüpft. Ebenso enthalten auch alle der folgenden Konstrukte, die an bestimmten Positionen eine Folge von Formen zulassen, ein implizites PROGN.

Eine auch aus anderen Programmiersprachen vertraute Kontrollstruktur ist die *If–Then–*[*Else*]–Anweisung:

IF *Test* *Then* [*Else*] [Special Form]
Zunächst wird die *Test*-Form evaluiert. Wenn das Ergebnis nicht NIL ist, wird die *Then*-Form evaluiert; sonst (sofern vorhanden) die *Else*-Form. Der Wert der evaluierten Form wird als Wert der IF-Form zurückgegeben.

WHEN *Test* {*Form*}* [Makro]
Zunächst wird die *Test*-Form evaluiert. Liefert sie einen anderen Wert als NIL, werden die Formen ausgeführt und der Wert der letzten Form als Wert der WHEN-Form zurückgeliefert; sonst evaluiert sie zu NIL.

UNLESS *Test* {*Form*}* [Makro]
Zunächst wird die *Test*-Form evaluiert. Liefert sie NIL, werden die Formen ausgeführt und der Wert der letzten Form als Wert der UNLESS-Form zurückgeliefert; sonst evaluiert sie zu NIL.

WHEN und UNLESS sind Makros, die auf der *Special Form* IF basieren. Bei der Evaluierung werden Formen, die diese Makros enthalten in korrespondierende IF-Formen expandiert. Es gilt:

(IF <*Test*> <*Then*-Form> NIL) ≡ (WHEN <*Test*> <*Then*-Form>) und
(IF <*Test*> NIL <*Else*-Form>) ≡ (UNLESS <*Test*> <*Else*-Form>).

Ein entscheidenender Unterschied zwischen einer IF-Form und einer WHEN-/UNLESS-Form besteht darin, daß in letzterer die auszuführende Aktion aus einer *Folge* von Formen bestehen kann, in der IF-Form dagegen nur aus einer einzelnen Form.

Die Standardkontrollstruktur in LISP ist das COND(itional), das aus einer Folge von Bedingungs–Aktionspaaren besteht:

COND	{(*Test* {*Form*}*)}*	[Makro]

Eine COND-Form besteht aus einer Folge von Klauseln. Jede Klausel besteht aus einem Test und einer Folge von Formen. Die Tests der Klauseln werden nacheinander ausgewertet, bis eine Klausel gefunden wird, deren Test zu einem anderen Wert als NIL evaluiert. Darauf werden die Formen dieser Klausel evaluiert und der Wert der letzten Form wird als Wert der COND-Form zurückgegeben. Gibt es keine derartige Klausel, evaluiert die COND-Form zu NIL.

Die Syntax einer COND-Form läßt sich durch folgendes Schema darstellen:

```
(cond ( <test 1> {<Aktion>}* )      ; 1. Klausel
      ( <test 2> {<Aktion>}* )      ; 2. Klausel
        ⋮          ⋮                    ⋮
      ( <test n> {<Aktion>}* ))     ; n-te Klausel
```

IF-, WHEN- und UNLESS-Formen lassen sich als Spezialfälle von COND-Formen auffassen; d.h. sie lassen sich sehr einfach als COND-Formen reformulieren.

In vielen Fällen ist es sinnvoll, als Test der letzten Klausel T zu verwenden, um sicherzustellen, daß die COND-Form nicht einfach zu NIL evaluiert. Ein Test dieser Form bildet eine sogenannte *Catch-all*–Bedingung und kann gelesen werden als: „*in allen anderen Fällen führe die folgenden Aktionen aus:*“. Werden in einer Klausel keine Aktionen angegeben, dann evaluiert die COND-Form, wenn die Evaluierung des Tests einen anderen Wert als NIL liefert, zu dem Wert dieses Tests.

Neben dem *Conditional* gibt es noch das CASE-Makro:

CASE	*Key-Form* {({({*Key*}*) | *Key*} {*Form*}*)}*	[Makro]

Die CASE-FORM besteht aus einer *Key-Form* und einer Folge von Klauseln. Jede Klausel besteht aus einem *Key* bzw. einer Liste von *Keys* und einer Folge von Formen.
Zunächst wird die *Key-Form* evaluiert. Dann wird die erste Klausel gesucht, deren *Key* mit dem Wert der *Key-Form* identisch ist bzw. in deren *Key*-Liste der Wert der *Key-Form* enthalten ist. Anschließend werden die Formen dieser Klausel ausgeführt und als Wert der CASE-Form der Wert der letzten Form zurückgegeben. Gibt es keine derartige Klausel, evaluiert die CASE-Form zu NIL.

Als *Key* der letzten Klausel kann auch T verwendet werden. Diese Klausel bildet dann die *Catch-all*–Klausel; d.h. der Vergleich mit diesem *Key* gelingt immer.

Welche Konsequenzen die Verwendung der verschiedenen Kontrollstrukturen auf die Definition einer Funktion hat, demonstriert das folgende Beispiel:

Beispiel (3-4)

```
;;; FUNKTION      :  ANFANG-&-ENDE1
;;; ARGUMENT(e)   :  Eine beliebige Liste.
;;; WERT          :  Eine Liste, die das erste und letzte Element der
;;;                  Argumentliste enthält gdw. sie nicht leer ist;
;;;                  sonst NIL.

(defun anfang-&-ende1 (liste)
  (if (endp liste)
      liste
      (cons (first liste) (last liste))))
```

Soll anstelle einer IF-Form eine WHEN- bzw. UNLESS-Form verwendet werden, muß man die Tatsache ausnutzen, daß diese Formen zu NIL evaluieren, wenn der in ihnen enthaltene Test zu NIL evaluiert:

```
(defun anfang-&-ende2 (liste)
  (when (not (endp liste))              ; Ist LISTE leer, dann evaluiert
    (cons (first liste) (last liste))))  ; die WHEN-Form zu NIL.
(defun anfang-&-ende3 (liste)
  (unless (endp liste)                  ; Ist LISTE nicht leer, evaluiert
    (cons (first liste) (last liste))))  ; die UNLESS-Form zu NIL.
```

Die Verwendung von COND und CASE ermöglichen zahlreiche Variationen, von denen wir nur zwei vorstellen. Charakteristisch für ANFANG-&-ENDE4 bzw. ANFANG-&-ENDE5 ist, daß sie drei Fällen unterscheiden:

1. die Argumentliste ist leer oder enthält genau zwei Elemente;
2. die Argumentliste enthält ein Element;
3. die Argumentliste enthält mehr als zwei Elemente.

```
(defun anfang-&-ende4 (liste)
  (cond ((or (endp liste) (= (length liste 2))) liste)
        ((= (length liste 1)) (cons (first liste) liste))
        (t (cons (first liste) (last liste)))))

(defun anfang-&-ende5 (liste)
```

```
(case (length liste)
      ((0 2) liste)
      (1 (cons (first liste) liste))
      (t (cons (first liste) (last liste)))))
```

3.3 Zeichenspiele

Zum Abschluß dieses Kapitels demonstrieren wir anhand eines größeren Beispiels, wie ein Programm aufgebaut werden sollte und wie die in diesem Kapitel vorgestellten Kontrollstrukturen sinnvoll verwendet werden können.

Als Ausgangsproblem wählen wir die Entwicklung eines "Wortgenerators"; d.h eines Progamms, das auf Grundlage von Mustern der Form *<konsonant vokal>, <vokal konsonant konsonant>*, ... etc. diesen Mustern korrespondierende Zeichenfolgen generiert. Wir gehen bei der Programmentwicklung von folgenden Annahmen aus:

1. Die generierten *Worte* werden durch Listen von Zeichen repräsentiert.
2. Es werden nur Muster bis zur Länge 3 behandelt.
3. Es werden vier Typen von Zeichen unterschieden:
   - Vokale
   - Konsonanten
   - Satzzeichen
   - Sonderzeichen

3.3.1 Deklarationen

Zunächst deklarieren wir die globalen Variablen, die wir zur Repräsentation der drei Typen von Zeichen, die wir unterscheiden wollen, verwenden werden:

```
(defvar *vokale*
        '(#\a #\e #\i #\o #\u))
(defvar *konsonanten*
        '(#\b #\c #\d #\f #\g #\h #\j #\k #\l #\m #\n #\p
          #\q #\r #\s #\t #\v #\w #\x #\y #\z))
(defvar *satzzeichen*
        '(#\. #\, #\; #\: #\! #\?))
```

Da die verfügbaren Sonderzeichen von dem zugrundeliegenden Zeichensatz abhängig sind und ihre Zahl sehr groß sein kann, werden sie nicht explizit spezifiziert. Jedes Zeichen, das kein Vokal, Konsonant oder Satzzeichen ist, wird als Sonderzeichen klassifiziert.

3.3.2 Prädikate

Im nächsten Schritt definieren wir für die von uns festgelegten Klassen von Objekten geeignete Typprädikate:

```
(defun konsonant-p (zeichen)
  (if (member zeichen *konsonanten*) t nil))

(defun vokal-p (zeichen)
  (when (member zeichen *vokale*) t))

(defun satzzeichen-p (zeichen)
  (when (member zeichen *satzzeichen*) t))

(defun sonderzeichen-p (zeichen)
  (and (not (konsonant-p zeichen))
       (not (vokal-p zeichen))
       (not (satzzeichen-p zeichen))))
```

Die zuvor definierten Typprädikate können nun verwendet werden, um Funktion ZEICHEN-TYP zu definieren, die für ein beliebiges Zeichen den Typ dieses Zeichens bestimmt:

```
(defun zeichen-typ (zeichen)
  (cond ((not (charakterp zeichen)) 'unzulaessiges_Objekt!)
        ((vokal-p zeichen) 'vokal)
        ((konsonant-p zeichen) 'konsonant)
        ((satzzeichen-p zeichen) 'satzzeichen)
        (t 'sonderzeichen)))
```

Beispiel (3-5)

```
(konsonant-p #\.)      ⟹  NIL
(konsonant-p #\c)      ⟹  T
(vokal-p #\z)          ⟹  NIL
(vokal-p #\e)          ⟹  T
(satzzeichen-p #\f)    ⟹  NIL
(satzzeichen-p #\.)    ⟹  T
(sonderzeichen-p #\a)  ⟹  NIL
(sonderzeichen-p #\%)  ⟹  T
(zeichen-typ #\a)      ⟹  VOKAL
(zeichen-typ #\l)      ⟹  KONSONANT
(zeichen-typ #\?)      ⟹  SATZZEICHEN
(zeichen-typ #\#)      ⟹  SONDERZEICHEN
(zeichen-typ 'a)       ⟹  'UNZULAESSIGES_OBJEKT!
```

3.3.3 Funktionen

Nun können wir mit der Entwicklung des eigentlichen Programms beginnen, das wir *top-down* entwerfen werden:

```
;;; FUNKTION      :  WORT-GENERATOR
;;; ARGUMENT(e)  :  Eine Liste mit Schlüsselworten.
;;; WERT         :  Eine Zeichenliste bzw. eine Fehlermeldung,
;;;                 falls die Argumentliste mehr als drei Elemente
;;;                 enthält.
;;;                 Top-Level Funktion des Programms.

(defun wort-generator (muster)
  (cond ((endp muster) nil)                    ; Ein leeres Muster.
        ((> (length muster) 3)                 ; Es werden nur Muster bis
         'unzulaessige-musterlaenge)           ; zur Länge 3 akzeptiert.
        (t (uebersetze-muster muster))))       ; Ein legales Muster wird übersetzt.

;;; FUNKTION      :  UEBERSETZE-MUSTER
;;; ARGUMENT(e)  :  Eine Musterliste legaler Länge.
;;; WERT         :  Eine Zeichenliste.
;;;                 Diese Funktion erzeugt die Zeichenliste.

(defun uebersetze-muster (muster)
  (case (length muster)
        (1 (list (generiere-zeichen (first muster))))          ; 1-Zeichen Wort
        (2 (list (generiere-zeichen (first muster))            ; 2-Zeichen Wort
                 (generiere-zeichen (second muster))))
        (3 (list (generiere-zeichen (first muster))            ; 3-Zeichen Wort
                 (generiere-zeichen (second muster))
                 (generiere-zeichen (third muster))))
        (t 'irgendetwas-ist-schiefgegangen)))                  ; Sollte nie eintreten

;;; FUNKTION      :  GENERIERE-ZEICHEN
;;; ARGUMENT(e)  :  Ein Schlüsselwort.
;;; WERT         :  Ein Zeichen bzw. ***.
;;;                 Übersetzt ein Schlüsselwort in ein passendes Zei-
;;;                 chen.

(defun generiere-zeichen (ausdruck)[3]
  (cond ((eq ausdruck 'vokal)
         (waehle-zeichen *vokale*))
        ((eq ausdruck 'konsonant)
         (waehle-zeichen *konsonanten*))
```

```
        ((eq ausdruck 'satzzeichen)
         (waehle-zeichen *satzzeichen*))
        (t (list '***))))

;;; FUNKTION      :  WAEHLE-ZEICHEN
;;; ARGUMENT(e)   :  Eine Zeichenmenge.
;;; WERT          :  Eine Zeichen aus dieser Zeichenmenge.
;;;                  Wählt per Zufallszahl ein Zeichen aus der Zei-
;;;                  chenmenge aus.

(defun waehle-zeichen (menge)
  (list (nth (random (length menge)) menge)))
```

Beispiel (3-6)

```
> (wort-generator '(vokal konsonant satzzeichen))
  (#\u #\m #\?)
> (wort-generator '(vokal konsonant konsonant))
  (#\u #\h #\f)
> (wort-generator '(vokal konsonant konsonant))
  (#\o #\c #\h)
> (wort-generator '(vokal sonderzeichen konsonant))
  (#\o *** #\m)
```

[3] Auf diese Funktion könnte verzichtet werden, wenn die globalen Variablen selbst zur Konstruktion der Muster verwendet würden; d.h. wenn WORT-GENERATOR z.B. mit '(*vokale* *vokale* *konsonanten*) aufgerufen würde.

Kapitel 4

Variablen

In diesem Kapitel werden wir uns ausführlich mit verschiedenen Typen von Variablen und den beiden in Common LISP zur Verfügung stehenden Bindungsstrategien auseinandersetzen. Außerdem werden wir zeigen, wie sich Funktionen innerhalb anderer Funktionen definieren lassen (*lokale Funktionen*) und Funktionen mit funktionalen Argumenten und funktionalen Werten (*Funktionen höherer Ordnung*) kennenlernen.

4.1 Variablentypen und Bindungsprinzipien

Variablen sind in den ersten beiden Kapiteln nur am Rande behandelt worden. Wir wissen bislang nur, daß Symbole u.a. die Funktion von Variablen übernehmen können: So kann ihnen mit SETQ ein Wert (Wertbindung) zugewiesen werden. Wir wissen außerdem, wie bei einem Funktionsaufruf die durch die Symbole der Lambda-Liste der Funktion bezeichneten Variablen (formale Parameter) an die Werte der Argumentbezeichner (aktuelle Parameter) gebunden werden. Ein wichtiger Unterschied zwischen diesen beiden Verwendungen von Symbolen besteht darin, daß im ersten Fall (außer bei Verschattung) die etablierte Wertbindung globale Geltung besitzt (*globale* Variablen), im zweiten Fall dagegen nur innerhalb der Funktion besteht ((*lokale* Variablen). Damit ist das Problem des Geltungsbereiches von Variablen angesprochen, das sich als Frage so formulieren läßt: Welche Bedingungen stellen es sicher/schließen es aus, daß ein Programmabschnitt auf eine zuvor generierte Variable zugreifen darf?

Geltungsbereich.
Der Geltungsbereich einer Variablen v ist der Bereich eines Programms, innerhalb dessen sie referenzierbar ist; d.h. innerhalb dessen die Evaluierung des sie bezeichnenden Symbols nicht zu einer Fehlermeldung der Form "unbound variable ..." führt. In Common LISP gibt es zwei Bindungsstrategien, durch die der Geltungsbereich von Variablen bestimmt wird und die als *lexikalische*

Bindung und *dynamische Bindung* bezeichnet werden. Auf beide Bindungsstrategien werden wir im folgenden ausführlich eingehen.

4.1.1 Lokale Variablen

Der Geltungsbereich von Variablen, die in der Lambda-Liste einer Funktion generiert werden, ist in Common LISP, anders als in älteren LISP-Dialekten, der Funktionskörper dieser Funktion; d.h. sie sind außerhalb der Funktion nicht referenzierbar. Variablen dieses Typs werden üblicherweise als *lokale Variablen* bezeichnet.

Beispiel (4-1)

```
> (defun raus-wie-rein (argument) argument)
  RAUS-WIE-REIN
> (raus-wie-rein "Hallo")
  "Hallo"
> argument
  "Error: Unbound Variable ARGUMENT"
1>
```

Der formale Parameter ARGUMENT wird an das Argument "Hallo" gebunden. Mit der Terminierung des Funktionsaufrufs wird diese Bindung wieder aufgehoben.

Erzeugung lokaler Variablen

Die Variablen der Lambda-Liste einer Funktion bilden, anschaulich formuliert, eine Schnittstelle zwischen der *Außenwelt* (Programmumgebung) und der *Innenwelt* der Funktion (Anweisungsblock), über die der Funktion die Werte übermittelt werden, die sie zur Durchführung ihrer Berechnungen benötigt.

In vielen Fällen ist es wünschenswert, über weitere lokale Variablen zu verfügen, die zur Speicherung von Zwischenergebnissen verwendet werden können und es erlauben, die mehrfache Ausführung eines Berechnungsschrittes zu vermeiden. Solche zusätzlichen lokalen Variablen können mit **LET** bzw. **LET*** erzeugt werden[1]. Für diese Variablen kann ein Anfangswert festgelegt werden, der durch eine beliebige LISP-Form spezifiziert werden kann. Variablen mit und ohne Anfangswert können beliebig miteinander kombiniert werden.

LET	({*Var* | (*Var Anfangswert*)}*)	{*Form*}*	[Special Form]

Das erste Argument einer LET-Form spezifiziert die innerhalb dieser Form gültigen lokalen Variablen. LET erzeugt diese Variablen und weist ihnen NIL bzw. den angegebenen Anfangswert zu. Diese Wertzuweisung erfolgt *parallel*. Anschließend werden die Formen des Anweisungsblocks evaluiert und der Wert der letzten Form als Wert der LET-Form zurückgegeben. Der Geltungsbreich der in der LET-Form generierten Variablen beschränkt sich auf die LET-Form.

LET*	({*Var* | (*Var Anfangswert*)}*)	{*Form*}*	[Special Form]

Der Unterschied zu LET besteht ausschließlich darin, daß die in LET* generierten Variablen *sequentiell* gebunden werden; d.h. die Form mit der der Anfangswert einer Variablen v_j spezifiziert wird, kann auf den Wert der Variablen v_{i} ($\mathrm{i} < \mathrm{j}$) referieren.

Die Verwendung zusätzlicher lokaler Variablen kann sinnvoll sein, wenn durch sie die Definition einer Funktion an transparenz gewinnt oder sie es zu vermeiden erlauben, daß eine möglicherweise komplexe Berechnung mehrfach ausgeführt werden muß:

Beispiel (4-2)

```
;;; FUNKTION      :  ANFANG-&-ENDE
;;; ARGUMENT(e)   :  Eine beliebige nicht-leere Liste.
;;; WERT          :  Eine Liste, die das erste und letzte Element der
;;;                  Argumentliste enthält. VORSICHT: Bei Listen
;;;                  mit weniger als zwei Elementen liefert die Funk-
;;;                  tion unplausible Resultate.

(defun anfang-&-ende (liste)
  (let ((anfang (first liste))      ; Berechne das erste Element der Liste.
        (ende (last liste)))        ; Berechne das letzte Element der Liste.
    (cons anfang ende)))
```

[1] Andere Formen, die die Generierung zusätzlicher lokaler Variablen erlauben, wie z.B. DO, DO*, PROG, (...), werden in den folgenden Kapiteln eingeführt.

```
;;; FUNKTION       :  WAS-SOLLS1
;;; ARGUMENT(e)    :  Eine beliebige Liste.
;;; WERT           :  Eine Liste, die zwei Listen enthält: Die erste Li-
;;;                   ste enthält das erste und zweite Element und
;;;                   die zweite das erste und letzte Element der Ar-
;;;                   gumentliste.

(defun kungelei1 (liste)
  (list (list (first liste) (second liste))
        (cons (first liste) (last liste))))
```

Der wegen der geringen Komplexität der Anweisung (FIRST LISTE) zwar vernachlässigbare Nachteil dieser Version besteht darin, das diese Anweisung zweimal berechnet wird. Die folgende Version der Funktion vermeidet solche Mehrfachberechnungen:

```
(defun kungelei2 (liste)
  (let ((anfang (first liste)))          ; Berechne erstes Element der Liste.
    (list (cons anfang (list (second liste)))
          (cons anfang (last liste)))))
```

Der einzige Unterschied zwischen LET- und LET*-Formen besteht darin, wann die in ihnen deklarierten Variabeln gebunden werden. Da LET die Variablen *parallel* bindet, ist x zu dem Zeitpunkt, zu dem der Wert von y berechnet wird, nicht definiert. In diesen Fällen, in denen auf den Wert einer voranstehenden Variablen zugegriffen werden soll, ist LET* zu verwenden:

Beispiel (4-3)

```
(defun spielerei1 (zahl)
  (let ((x zahl)
        (y (* zahl 2)))
       (* x y)))
(defun spielerei2 (zahl)
  (let ((x zahl)
        (y (* 2 x)))
       (* x y)))
> (spielerei1 3)
  18
> (spielerei2 3)
  "Error: Unbound Variable X"
(defun spielerei3 (zahl)
```

```
  (let* ((x zahl)
         (y (* x 2)))
        (* x y)))
> (spielerei3 3)
  18
```

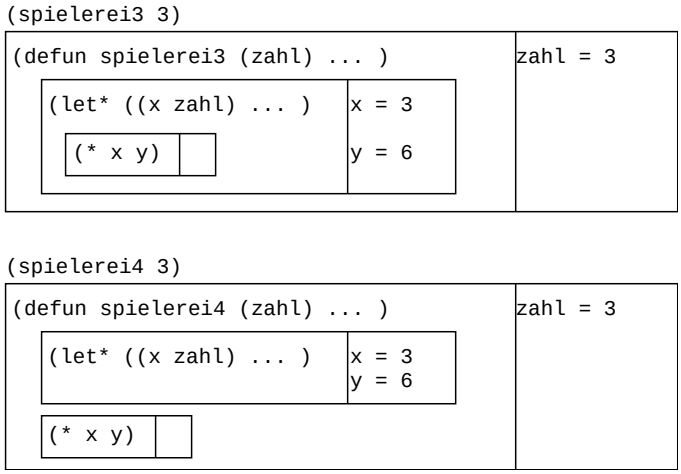
Wichtig zu beachten ist auch, daß zwar die Variablen aus der Lambda-Liste der Funktion innerhalb einer eingebetteten LET/ LET*-Form referenzierbar sind (s.o.), nicht aber umgekehrt die innerhalb der LET/LET*-Form generierten Variablen im übrigen Funktionskörper:

Beispiel (4-4)

```
> (defun spielerei4 (zahl)
    (let ((x zahl)
          (y (* zahl 2))))
        (* x y))
  SPIELEREI4
> (spielerei4 3)
  "Error: Unbound Variable X"[2]
```

Um zu verdeutlichen, warum die vierte Version von SPIELEREI anders als die dritte eine Fehlermeldung produziert, betrachten wir die beim Funktionsaufruf generierten Umgebungen:

[2]Um an eine bekannte terminologische Unterscheidung zu erinnern: Alle in der Lambda-Liste einer Funktion generierten Variablen sind innerhalb des Funktionskörpers *gebunden*; ebenso sind alle in einer LET/ LET*-Form generierten Variablen in dieser Form gebunden. Alle in einer Funktion verwendeten Variablen, die nicht in der Lambda-Liste generiert werden oder außerhalb der sie generierenden LET/LET*-Form vorkommen, werden als *freie Variablen* dieser Funktion bezeichnet. In SPIELEREI4 kommen "x" und "y" im Gegensatz zu den vorangegangenen Versionen *frei* vor.



Zu einer Umgebung gehören nur die Bindungen der sie umschließenden Umgebungen: Wenn (* x y) beim Aufruf von SPIELEREI4 evaluiert wird, ist die zu diesem Zeitpunkt aktuelle Umgebung in keiner Umgebung enthalten, in der die durch "x" bzw. "y" bezeichneten Variablen gebunden sind.

Verschattung.
Ein anderes, aus anderen Programmiersprachen bekanntes Problem ergibt sich, wenn es in einer Umgebung U eine Variable v gibt, die denselben Namen n hat, wie eine Variable v' einer U umschließenden Umgebung: Auf welche Variable referiert n, wenn eine Form in der Umgebung U evaluiert wird? Die Suche nach der Antwort auf diese Frage sollte kein Kopfzerbrechen bereiten: In diesem Fall evaluiert n zum Wert von v. Die Variable v *verschattet* die Variable v'.

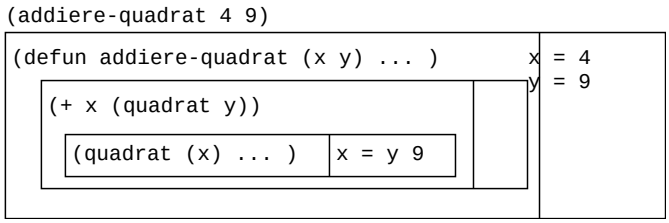
Beispiel (4-5)

```
> (let ((a 3) (b 4))
    (let ((a 5) (c 6))          ; Das äußere "a" wird durch
      (* a b c)))               ; das innere verschattet.

  120
> (defun addiere-quadrat (x y)
    (+ x (quadrat y)))
  ADDIERE-QUADRAT
> (defun quadrat (x)
    (* x x))
  QUADRAT
> (addiere-quadrat 4 9)
```

85

Bei der Evaluierung des Aufrufs der Funktion Quadrat innerhalb dieses Funktionsaufrufs verschattet die innere Bindung von "x" die äußere:



4.1.2 Bindungsstrategien

In den Beispielen der vorangegangenen Abschnitte haben wir nur solche Funktionen betrachtet, die ausschließlich gebundene Variablen verwendeten. Es stellt sich die Frage, wie in Funktionen vorkommende freie Variablen gebunden werden bzw. unter welchen Bedingungen freie Variablen in Funktionen zulässig sind. In Common LISP können unterschiedliche Bindungsstrategien[3] verwendet werden:

Lexikalische Bindung

In Common LISP gilt, sofern nicht durch eine Deklaration explizit vereinbart, das Prinzip der *lexikalischen Bindung*: Der Geltungsbreich einer Variablen ist in diesem Fall identisch mit dem Programmbereich (im Sinne von Textabschnitt), innerhalb dessen sie definiert wurde; d.h. der Geltungsbereich einer in einer Funktion generierten Variablen wird zum Zeitpunkt der **Definition** der Funktion und nicht zum Zeitpunkt ihrer **Ausführung** determiniert.

Aufgrund dieses Prinzips ist der Geltungsbereich von Variablen, die in der Lambda-Liste einer Funktion generiert werden, der Anweisungsblock, der den Funktionskörper bildet; und der Versuch, außerhalb dieses Bereiches auf diese Variablen zu referieren, führt, wie wir gesehen haben, zu Fehlermeldungen. So wird auch verhindert, daß eine Funktion eine freie Variable enthalten kann, die durch eine sie aufrufende Funktion gebunden wird.

Beispiel (4-6)

```
> (defun uebergabe (argument)
    (rein-wie-raus))
  UEBERGABE
```

[3] Im Englischen spricht man von *scoping rules* und *lexical scoping* bzw. *dynamic scoping*; vgl. Steele[1990], S.42-46.

```
> (defun rein-wie-raus ()        ; REIN-WIE-RAUS wird als
    argument)                    ; parameterlose Funktion definiert.
  REIN-WIE-RAUS
> (uebergabe "Hallo")
  "Error: Unbound Variable ARGUMENT"
```
[4]

Dynamische Bindung

Das Prinzip der dynamischen Bindung, das besonders in älteren LISP-Dialekten verwendet wurde, unterscheidet sich von dem der lexikalischen Bindung darin, daß die formalen Parameter einer Funktion, nachdem sie beim Aufruf der Funktion gebunden wurden, so lange gebunden bleiben, bis die Evaluierung des Funktionsaufrufs mit der Rückgabe des Funktionswerts abgeschlossen wird. Wenn im vorangegangenen Beispiel ARGUMENT eine dynamisch gebundene Variable bezeichnet hätte, dann hätte die Evaluierung der Form (UEBERGABE "Hallo") nicht zu einer Fehlermeldung geführt.

Es ist in Common LISP möglich, neben lexikalisch gebundenen Variablen dynamisch gebundene Variablen, die auch *Special–Variablen* genannt werden, zu verwenden. Um innerhalb einer Funktion eine dynamisch gebundene Variable zu erzeugen, muß eine bereits generierte Variable als Special–Variable deklariert werden. Alle Funktionen, die auf diese Variable zugreifen müssen ebenfalls eine Special–Deklaration für diese Variable enthalten. Special–Deklarationen haben die Form:

(declare (special var*))[5]

Um bei dem Aufruf der in Beispiel (3-5) definierten Funktionen UEBERGABE bzw. REIN-WIE-RAUS eine Fehlermeldung zu vermeiden, sind folgende Deklarationen erforderlich:

Beispiel (4-7)

```
> (defun uebergabe (argument)
    (declare (special argument))
    (rein-wie-raus))
  UEBERGABE
> (defun rein-wie-raus ()
    (declare (special argument))
    argument)
  REIN-WIE-RAUS
> (uebergabe "Hallo")
```

[4] Durch das Prinzip der lexikalischen Bindung in Common LISP wird das sogen. "FunArg"-Problem vermieden; vgl. Kapitel 1, Fußnote 1.

[5] Da wir von Deklarationen nur selten Gebrauch machen werden, wird ihr Format hier nur kurz skizziert.

"Hallo"

Beide Deklarationen sind notwendig: Durch die erste Deklaration wird bestimmt, daß für den Zeitraum der Ausführung von UEBERGABE die Variable ARGUMENT dynamisch gebunden ist und damit innerhalb des Aufrufs von REIN-WIE-RAUS referenzierbar ist. Die zweite Deklaration legt fest, daß alle Vorkommen dieser Variable in REIN-WIE-RAUS dynamisch gebunden sind. Wichtig ist zu beachten, daß die Variable nur innerhalb dieser beiden Funktionen dynamisch gebunden ist.

Special–Deklarationen können in DEFUN und LET/LET*-Formen im Anschluß an die Variablenliste eingefügt werden. Grundsätzlich sollte man mit solchen Deklarationen sparsam umgehen: Sie erhöhen nicht gerade die Transparenz der Programme.

4.1.3 Globale Variablen

Es bleibt die Frage, wie die außerhalb von Funktionen mit SETQ generierten Variablen gebunden werden. Sie werden dynamisch gebunden und sind damit, sofern sie in der Null-Umgebung[6] generiert werden, in allen Funktionen - außer wenn sie verschattet werden - referenzierbar und übernehmen damit die Funktion von *globalen Variablen*. Damit gibt es also drei Typen von Variablen, auf die innerhalb einer Funktion Bezug genommen werden kann:

1. die *lokalen* Variablen, die innerhalb einer Funktion generiert wurden;

2. die Variablen, die in der sie aufrufenden Funktion und der aufgerufenen Funktion als Special-Variablen deklariert wurden, und

3. die Variablen, die in der Umgebung existieren, in der die Funktion definiert wurde.

Eine Verwendung von globalen Variablen ist dann sinnvoll, wenn es innerhalb eines Programms Informationen gibt, auf die diverse Funktionen des Programms zugreifen müssen, und es auch unter konzeptuellen Aspekten vertretbar erscheint (z.B. die Transparenz des Programms erhöht wird), diese Informationen so zu repräsentieren und nicht als Parameter bei Funktionsaufrufen weiterzugeben. Grundsätzlich aber sollte man globale Variablen sparsam verwenden und insbesondere die Zahl der Funktionen, die ihren Wert ändern können, gering halten, da sonst die Frage, welchen Wert eine globale Variable v zu einem Zeitpunkt t besitzt, nicht geringes Kopfzerbrechen bereiten kann.

[6] Mit *Null-Umgebung* wird die Umgebung bezeichnet, die nach Initialisierung des LISP-Systems aktiv ist. Diese Umgebung ist in keine andere Umgebung eingebettet und enthält alle während der Systembenutzung generierten Umgebungen.

Bislang haben wir in unseren Beispielen SETQ als *Konstruktor* und auch als *Modifikator* verwendet; d.h. als eine Funktion, die per Seiteneffekt globale Variablen generiert bzw. ihre Werte ändert. Da es in Common LISP spezielle Funktionen zum Deklarieren von globalen Variablen (und Konstanten) gibt, gilt es als schlechter Programmierstil, SETQ als Konstruktor zu verwenden.

DEFVAR	*Name* [*Anfangswert*]	[Makro]

DEFPARAMETER	*Name* *Anfangswert*	[Makro]

DEFCONSTANT	*Name* *Anfangswert*	[Makro]

Die mit diesen Funktionen generierten Variablen sind, wie schon erwähnt, in allen Funktionen referenzierbar. Der Unterschied zwischen DEFVAR und DEFPARAMETER besteht darin, daß erstens in DEFVAR-Formen die Angabe eines Anfangswertes optional ist, zweitens durch die Verwendung von DEFPARAMETER der Benutzer signalisiert, daß der generierten Variablen ein Systemparameter zugewiesen wurde, von dem nicht zu erwarten ist, daß er häufig geändert wird und drittens mit DEFPARAMETER Variablen redefiniert werden können. Der Versuch, einer mit DEFCONSTANT generierten Variablen einen neuen Wert zuzuweisen, führt zu einer Fehlermeldung.

Beispiel (4-8)

```
> (defvar *auto* 'ford)
  *AUTO*
> *AUTO*
  FORD
> (defvar *auto* 'vw)
  *AUTO*
> *auto*
  FORD
```

aber:

```
> (defparameter *seitenlaenge* 40)
  *SEITENLAENGE*
> *seitenlaenge*
  40
```

```
> (defparameter *seitenlaenge* 60)
  *SEITENLAENGE*
> *seitenlaenge*
  60
```

4.2 Lokale Funktionsdefinitionen

Neben lokalen Variablen lassen sich in Common LISP auch lokale *Funktionen* definieren; d.h. Funktionen, die innerhalb einer anderen Funktion definiert werden und nur innerhalb dieser Funktion verwendet werden können.

Sinnvoll ist die Verwendung von lokalen Funktionsdefinitionen, wenn sicherzustellen ist, daß die so definierte Funktion von keiner anderen Funktion aufgerufen werden kann als der Funktion, innerhalb der sie definiert wurde, oder wenn es sich bei ihr um eine Hilfsfunktion handelt, die ausschließlich in diesem Kontext benötigt wird. Die einfachste Möglichkeit zur Generierung von lokalen Funktionen besteht in der Verwendung von Lambda-Ausdrücken.

Beispiel (4-9)

```
;;; FUNKTION      :  MEHRFACH-P
;;; ARGUMENT(e)  :  Ein Atom und eine Liste.
;;; WERT          :  T bzw. NIL.
;;;                  Die Funktion prüft, ob Atom mehrfach als Top-
;;;                  Level Element in Liste vorkommt.

> (defun mehrfach-p (atom liste)
    (declare (special liste))
    (if[7]((lambda (element)          ; Hier wird eine anonyme lokale
             (declare (special liste))  ; Funktion definiert.
             (member element (rest (member element liste))))
           atom)                      ; Argument der lokalen Funktion
          t                           ; then-Fall
          nil))                       ; else-Fall
  MEHRFACH-P
> (mehrfach-p 'a '(a b c ))
  NIL
> (mehrfach-p 'a '(a b c a))
  T
```

Die Bezeichnung von lokalen Funktionen durch Lambda-Ausdrücke ist umständlich, wenn die lokale Funktion mehrfach in der Funktion verwendet werden soll, innerhalb der sie definiert wurde, und es ist unmöglich, *rekursive* lokale Funktionen so zu definieren: Wie kann eine namenlose Funktion sich selbst aufrufen? Zur Generierung benannter lokaler Funktionen stehen die beiden *Special Forms* **LABELS** und **FLET** zur Verfügung:

FLET ({(*Name Lambda-Liste* {*Form*}*)}*) {*Form*}* [Special Form]
In einer FLET-Form kann eine beliebige Anzahl von lokalen Funktionen definiert werden. Eine Definition besteht (vgl. DEFUN-Form) aus *Funktionsname*, *Lambda-Liste* und dem den Funktionskörper bildenden *Anweisungsblock*. Der Geltungsbereich der so definierten Funktionen ist die FLET-Form, innerhalb der sie definiert wurden.

[7] Die Special-Form IF hat die Syntax:

(if *Bedingung then-Anweisung else-Anweisung*).

LABELS ({(*Name Lambda-Liste* {*Form*}*)}*) { *Form*}* [Special Form]

Die syntaktische Struktur von LABELS-Formen ist mit der der FLET-Formen identisch. Der einzige Unterschied zwischen LABELS und FLET besteht darin, daß mit LABELS rekursive Funktionen und sich wechselseitig aufrufende Funktionen definiert werden können: Der Geltungsbereich eines Funktionsnamens in einer LABELS-Form umfasst neben dem Anweisungsblock der LABELS-Form auch die in der Form enthaltenen Funktionsdefinitionen.

Beispiel (4-10)

```
;;; FUNKTION      :  WIE-OFT
;;; ARGUMENT(e)   :  Ein Atom und eine Liste.
;;; WERT          :  Die Zahl der Vorkommen von Atom in Liste.
;;;                  Die lokale Funktion REDUZIERE tilgt aus Liste
;;;                  alle Vorkommen anderer Ausdrücke.

> (defun wie-oft (symbol liste)
    (labels
     ((reduziere (l)              ; Begin der Definition der lokalen
        (if  (endp[8] l)          ; Funktion REDUZIERE
             ()
             (if (eq[9] (first l) symbol)
                 (cons (first l) (reduziere (rest l)))
                 (reduziere (rest l))))))      ; Ende der Definition.
      (length (reduziere liste))))             ; Aufruf der lokalen Funktion.
  WIE-OFT
> (wie-oft 'a '(b a a c))
  2
> (wie-oft 'd '(b a a c))
  0
```

[8] Das Prädikat ENDP prüft, ob eine gegebene Liste leer ist; d.h. es liefert T gdw. sein Argument zu NIL evaluiert.

[9] EQ ist eine Indentitätsfunktion: Sie liefert den Wert T gdw. beide miteinander verglichenen Objekte identisch sind, und NIL in allen anderen Fällen.

4.3 Funktionen mit funktionalen Argumenten

Wir haben bislang Funktionen betrachtet, die Zahlen, Symbole und Listen als Argumente nehmen. Tatsächlich gibt es in LISP kaum Restriktionen bezüglich des Argument- und Wertebereichs von Funktionen; d.h. es ist z.B. möglich, Funktionen höherer Stufe zu definieren, die als Argumente Funktionen nehmen. So könnte es wünschenswert erscheinen, über eine Funktion zu verfügen, die zwei Argumente - eine Funktion und eine Liste - akzeptiert, diese Funktion auf alle Top-Level Ausdrücke der Liste anwendet und als Wert die Liste der aus der Funktionsanwendung resultierenden Werte liefert.

Beispiel (4-11)

```
> (anwenden 'first '((a) (b) (c)))
  (A B C)
> (anwenden 'rest '((a) (b) (c)))
  (NIL NIL NIL)

;;; Ein naheliegender Versuch, ANWENDEN zu definieren, könnte
;;; folgende Form annehmen:
(defun anwenden (funktion argumentliste)
  (if (endp argumentliste)
      ()
      (cons (funktion (first argumentliste))
            (anwenden funktion (rest argumentliste)))))
;;; Allerdings arbeitet diese Version von ANWENDEN nicht wie
;;; gewünscht:
> (anwenden 'first '((a) (b) (c)))
  "Error - unknown Function FUNKTION"
```

Zu dieser Fehlermeldung kommt es, da beim Aufruf einer Funktion die Parameterübergabe eine Wertbindung und keine Funktionsbindung etabliert; d.h. bei der Ausführung von (FUNKTION (FIRST ARGUMENTLISTE)) existiert für den Parameter FUNKTION keine Funktionsbindung. Glücklicherweise gibt es eine vordefinierte LISP-Funktion, die uns hier weiterhelfen kann:

FUNCALL	*Funktion* *Argument**	[Funktion]
Bei Evaluierung einer FUNCALL-Form wird *funktion* auf die in der Form spezifizierten Argumente angewendet. Der Unterschied zu einem normalen Funktionsaufruf besteht darin, daß in den Fällen, in denen die Funktion durch ein Symbol bezeichnet wird, auf die *Wertbindung* des Symbols zugegriffen wird.		

Die korrekte Fassung von Anwenden lautet also:

```
(defun anwenden (funktion argumentliste)
  (if (endp argumentliste)
      ()
      (cons (funcall funktion (first argumentliste))
            (anwenden funktion (rest argumentliste)))))
```

Während FUNCALL erlaubt, auf die als Wertbindung eines Symbols gespeicherte Funktion zuzugreifen, in denen normale LISP-Evaluierung die Funktionsbindung des Symbols verwenden würde, stellt die Funktion FUNCTION sicher, daß auch dann auf die Funktionsbindung des Symbols zugegriffen wird, wenn das Symbol nicht in Funktionsposition - als erstes Element einer Liste - vorkommt: FUNCTION überführt einen Funktionsbezeichner in das durch ihn bezeichnete Funktionsobjekt.

FUNCTION	*Funktionsbezeichner*	[Special Form]
Als Wert eines FUNCTION-Aufrufs wird das durch *Funktionsbezeichner* bezeichnete Funktionsobjekt zurückgegeben: Ist der Funktionsbezeichner ein Symbol, evaluiert es zu seiner Funktionsbindung. Ist es ein Lambda-Ausdruck, wird eine *Lexical-Closure* erzeugt, und die durch sie repräsentierte Funktion wird unter Beachtung der für lexikalische Bindung geltenden Regeln ausgeführt.		

Statt FUNCTION läßt sich auch das Makro-Zeichen #' verwenden; d.h. es gilt:

$$\text{(function symbol)} \equiv \#\text{'symbol}$$

Eine weitere nützliche vordefinierte Funktion zweiter Stufe ist die Funktion APPLY, die es erlaubt, eine andere Funktion auf eine beliebige Zahl von Argumenten anzuwenden:

APPLY	*Funktion* *Argument*$^+$	[Funktion]
Bei der Evaluierung einer APPLY-Form wird *Funktion* auf die in der Form spezifizierten Argumente angewendet. Die Funktion kann durch einen Lambda-Ausdruck oder ein Symbol bezeichnet werden. Ist letzteres der Fall, wird die Funktionsbindung des Symbols verwendet.		

Beispiel (4-12)

```
> (apply '+ '(1 2 3))
  6
```

```
> (apply #'+ '(1 2 3))
  6
> (apply 'cons '((+ 2 3) 4))
  ((+ 2 3) . 4)
```

Weitere Funktionen zweiter Stufe (MAP...-Funktionen) werden wir kennenlernen, wenn wir uns mit iterativer Progammierung in LISP beschäftigen.

4.4 Funktionen mit funktionalen Werten

Neben Funktionen, die funktionale Argumente akzeptieren, ist es in LISP auch möglich, Funktionen zu definieren, die als Wert eine Funktion liefern. Wir werden in diesem Abschnitt eine Funktion entwickeln[10] (MAKE-POWER-OF-TWO-GENERATOR), die als Wert eine Generator-Funktion oder kurz einen *Generator* liefert, wobei wir unter einem Generator eine Funktion verstehen, die bei jedem Aufruf ein neues Objekt aus einer bestimmten Folge von Objekten als Wert liefert. Unsere Funktion wird Generatoren erzeugen, die ausgehend von 2^1 bei jedem Aufruf die nächst höhere Potenz von 2 als Wert liefern. Wenn POWER-OF-TWO ein solcher Generator ist, dann soll er sich wie folgt verhalten:

```
> (power-of-two)
  2
> (power-of-two)
  4
> (power-of-two)
  8
```

Offensichtlich kann der Generator auf den zuletzt erzeugten Wert zugreifen. Am einfachsten ist es, eine globale Variable zu deklarieren, auf die POWER-OF-TWO zugreifen kann:

```
> (defvar *previous-power-of-two* 1)
  *previous-power-of-two*
> (defun previous-power-of-two ()
    (setq *previous-power-of-two*
          (* *previous-power-of-two* 2)))
  PREVIOUS-POWER-OF-TWO
```

Ein Nachteil bei der Verwendung von globalen Variablen liegt darin, daß ihr Wert prinzipiell durch andere Funktionen verändert werden kann und daß die Definition mehrerer Generatoren eines Typs zu Konflikten führt bzw. die Einführung weiterer globaler Variablen

[10] Die in diesem Abschnitt verwendeten Beispiele stammen aus Winston&Horn[1989], S.216-24.

erzwingt. Wie kann ein Generator ohne Verwendung globaler Variablen definiert werden? Die Lösung dieser Frage lautet: durch *Verkapselung* der freien Variablen innerhalb eines Lambda-Ausdrucks. In einem ersten Schritt können wir unseren Generator auf folgende Weise definieren:

```
> (setf (symbol-function 'power-of-two)
  #'(lambda ()
      (setq *previous-power-of-two*
            (* *previous-power-of-two* 2))))
  #'(lambda () (setq *previous-power-of-two*
                     (* *previous-power-of-two* 2))))
```

Da durch die Verwendung von SYMBOL-FUNCTION die anonyme Funktion in das Funktionswert-Feld von POWER-OF-TWO eingetragen wurde, kann der Generator wie bei Verwendung einer DEFUN-Form ohne FUNCALL aufgerufen werden:

```
> (power-of-two)
  2
> (power-of-two)
  4
> (power-of-two)
  8
```

Im zweiten Schritt eleminieren wir die globale Variable PREVIOUS-POWER-OF-TWO und ersetzen sie durch eine lokale Variable gleichen Namens, indem wir den Lambda-Ausdruck in eine LET-Form einbetten:

```
> (setf (symbol-function 'power-of-two)
        (let ((previous-power-of-two 1))
          #'(lambda ()
              (setq previous-power-of-two
                    (* previous-power-of-two 2)))))
  <lexical-closure ...>
```

Bei jedem Aufruf des Generators wird auf die innerhalb der durch die LET-Form erzeugten Umgebung geltende Bindung der *lokalen Variable* PREVIOUS-POWER-OF-TWO zugegriffen und diese Bindung verändert; andere Funktionen können nicht mehr auf sie zugreifen bzw. sie verändern. Damit ist jetzt auch die Möglichkeit gegeben, verschiedene Generatoren dieses Typs zu generieren, ohne daß es beim Variablenzugriff zu Konflikten kommt.

```
> (setf (symbol-function 'generator1)
```

```
        (let ((previous-power-of-two 1))
          #'(lambda ()
              (setq previous-power-of-two
                    (* previous-power-of-two 2)))))
  <lexical-closure ...>
> (setf (symbol-function 'generator2)
        (let ((previous-power-of-two 1))
          #'(lambda ()
              (setq previous-power-of-two
                    (* previous-power-of-two 2)))))
  <lexical-closure ...>
> (generator1)
  2
> (generator1)
  4
> (generator2)
  2
> (generator1)
  8
```

Die so definierten Generatoren erzeugen bei jedem Aufruf die nächste Zweierpotenz. Es ist aufgrund der Verkapselung von PREVIOUS-POWER-OF-TWO nicht möglich, ihren Wert von *außen* zu beeinflussen, um sie z.B. auf ihren Anfangswert zurückzusetzen. Um einen Generator in seinen Ausgangszustand zurückzuversetzen, benötigen wir eine eigene *Reset*-Funktion. Wir können die Reset-Funktion wie die Generator-Funktion in die LET-Form einbetten - so ist sichergestellt, daß auch sie auf PREVIOUS-POWER-OF-TWO zugreifen kann.

```
> (setq procedures
        (let ((previous-power-of-two 1))
          (list #'(lambda ()
                    (setq previous-power-of-two 1))
                #'(lambda ()
                    (setq previous-power-of-two
                          (* previous-power-of-two 2))))))
  <lexical-closure ...>
```

Den Generator und die zugehörige Reset-Funktion erhalten wir durch:

```
(setf (symbol-function 'power-of-two-reset) (first procedures)
      (symbol-function 'power-of-two-value) (second procedures))
```

Der Generator wird mit "(power-of-two-value)" aufgerufen und mit "(power-of-two-reset)" in seinen Ausgangszustand versetzt. Eine elegantere Lösung erhalten wir allerdings, indem wir eine komplexe Funktion definieren, die abhängig von dem Argument, mit dem sie aufgerufen wird, entweder den Generator oder die Reset-Funktion als Wert liefert:

```
> (setf (symbol-function 'generator-with-dispatch-procedure)
        (let* ((previous-power-of-two 1)
               (reset-procedure
                 #'(lambda () (setq previous-power-of-two 1)))
               (value-procedure
                 #'(lambda ()
                     (setq previous-power-of-two
                           (* previous-power-of-two 2)))))

          #'(lambda (accessor)
              (if (eq 'reset accessor)
                  (funcall reset-procedure)
                  (funcall value-procedure)))))
  <lexical-closure ...>
```

Der Generator und die Reset-Funktion werden an die innerhalb der LET-Form generierten lokalen Variablen RESET-PROCEDURE und VALUE-PROCEDURE gebunden. Der Generator wird aufgerufen mit:

```
(generator-with-dispatch-procedure 'reset) bzw.
(generator-with-dispatch-procedure 'value).
```

Eine weitere Erleichterung bei der Generierung mehrerer Generatoren eines Typs bietet die Definition einer Generatoren generierenden Funktion:

```
> (defun make-power-of-two-generator ()
    (let* ((previous-power-of-two 1)
           (reset-procedure
             #'(lambda () (setq previous-power-of-two 1)))
           (value-procedure
             #'(lambda () (setq previous-power-of-two
                                (* previous-power-of-two 2)))))
      #'(lambda (accessor)
          (if (eq 'reset accessor)
              (funcall reset-procedure)
              (funcall value-procedure)))))
    MAKE-POWER-OF-TWO-GENERATOR
```

Generatoren lassen sich jetzt mit dieser Funktion sehr einfach definieren:

```
> (setf (symbol-function 'generator1) (make-power-of-two-generator)
        (symbol-function 'generator2) (make-power-of-two-generator))
<lexical-closure ...>
> (generator1 'value)
2
> (generator1 'value)
4
> (generator2 'value)
2
> (generator1 'reset)
1
> (generator2 'value)
4
> (generator1 'value)
2
```

Kapitel 5

Rekursion

Die Funktionen, die wir bislang definiert haben, zeichen sich dadurch aus, daß zur Berechnung des Funktionswertes eine Sequenz einfacher Operationen ausgeführt werden muß. Bei komplexeren Probleme ist es häufig erforderlich, eine Folge von Operationen mehrfach auszuführen, ohne daß es von vornherein möglich ist zu spezifizieren, wie oft sie auszuführen ist. Probleme dieses Typs lassen sich durch *rekursive* (Kapitel 5) oder *iterative* Funktionen (Kapitel 6) lösen.

5.1 Grundlagen rekursiver Programmierung

Für komplexe **Probleme** läßt sich, sofern sie überhaupt lösbar sind, prinzipiell eine rekursive, wie auch eine iterative Lösung finden. Kein Problem erzwingt eo ipso eine rekursive (iterative) Lösung. Häufig legt allerdings eine gegebene **Problembeschreibung** es nahe, eine rekursive (iterative) Lösung zu suchen. Ein Vorteil rekursiver **Problemlösungen** liegt in der Übersichtlichkeit und *Transparenz*: Sie bilden die Struktur des zugrundeliegenden Problems häufig direkter ab als ihre iterativen Gegenstücke. Iterative Lösungen dagegen sind, wie wir noch sehen werden, häufig *effizienter* und zwar sowohl hinsichtlich ihres Laufzeitverhaltens, wie auch des durch sie beanspruchten Speicherplatzes.

5.1.1 Struktur rekursiver Funktionen

Eine Problemlösung wird als *rekursiv* bezeichnet, wenn sie das Ausgangsproblem so in Teilprobleme zerlegt, daß eines dieser Teilprobleme eine Instanz des Ausgangsproblems ist.

Beispiel (5-1)
Die Funktion f zur Berechnung der Fakultät einer Zahl ist wie folgt definiert:

f(n) = 1	gdw. n = 0 und
f(n) = n * f(n - 1)	gdw. wenn n > 0.

Aus dieser Definition folgt, daß f nur für Zahlen ≥ 0 definiert ist.

In dieser Definition werden zwei Situationen unterschieden:

1. Ist n = 0, dann ist keine weitere Berechnung notwendig.
2. Ist n > 0, dann wird das Ausgangsproblem durch folgende zwei Teilprobleme ersetzt:
   (a) Berechne den Wert von f(n - 1).
   (b) Multipliziere n mit der Fakultät von n−1.

   Das erste Teilproblem ist, wie man sofort sieht, eine Instanz des Ausgangsproblems.

Die Definition illustriert deutlich das minimale Inventar einer rekursiven Definition. Eine rekursive Definition muß mindestens zwei Klauseln enthalten:

1. Eine Klausel, die sicherstellt, daß der Berechnungsprozeß nach endlich vielen Schritten terminiert (*Abbruchbedingung*).
2. Eine Klausel, die das Ausgangsproblem in einfachere Teilprobleme zerlegt: Mindestens eines der Teilprobleme muß eine Instanz des Ausgangsproblems sein (*rekursiver Aufruf*).

Die korrekte Formulierung dieser Klauseln ist von entscheidender Bedeutung, wenn es darum geht sicherzustellen, daß der Berechnungsprozeß nach endlich vielen Schritten terminiert: Das Fehlen einer Abbruchbedingung, die Verwendung einer fehlerhaften Abbruchbedingung oder ein rekursiver Aufruf, in dem der Parameter der Funktion unverändert übernommen wird, führen dazu, daß erst ein *Stack–Overflow* diesen Prozeß beendet.

Die Definition der Fakultätsfunktion läßt sich direkt in eine rekursive LISP-Funktion abbilden:

Version (1)

```
(defun fakultaet1 (zahl)
  (if (= zahl 0)                    ; Abbruchbedingung
```

```
    1
    (* zahl (fakultaet1 (- zahl 1)))))      ; Rekursiver Aufruf
```

Die Funktion FAKULTAET1 scheint die oben gegebene Definition adäquat abzubilden. Allerdings berücksichtigt sie einen Aspekt der zugrundeliegenden Definition nicht: die Fakultätsfunktion ist nur für Zahlen ≥ 0 definiert; FAKULTAET1 dagegen akzeptiert auch negative Zahlen, terminiert aber in diesen Fällen nicht. Auch wenn es nicht immer sinnvoll ist, explizit zu überprüfen, daß die Parameter einer Funktion *Randbedingungen* dieses Typs erfüllt, scheint hier eine Änderung der Definition sinnvoll. Eine geringfügige Änderung der Abbruchbedingung stellt sicher, daß die Funktion in jedem Fall terminiert:

Version (2)

```
(defun fakultaet2 (zahl)
  (if (<= zahl 0)
      1
      (* zahl (fakultaet2 (- zahl 1)))))
```

Obwohl diese Funktion korrekt arbeitet, bleibt zu kritisieren, daß sie nicht die Tatsache abbildet, daß die Fakultätfunktion für Zahlen kleiner als Null nicht definiert ist.

Version (3)

```
(defun fakultaet3 (zahl)
  (cond ((< zahl 0) "Fakultaet: unzulaessiges Argument")
        ((= zahl 0) 1)
        (t (* zahl (fakultaet3 (- zahl 1))))))
```

Unsere Funktion FAKULTAET3 bildet die von uns verwendete Definition der Fakultätsfunktion zwar korrekt ab, ist aber ineffizient: bei jedem rekursiven Aufruf wird erneut überprüft, ob die als Parameter übergebene Zahl kleiner als Null ist, obwohl dieser Test nur einmal, nach dem ersten Aufruf der Funktion, ausgeführt werden muß, da die in der Funktion verwendeten Operationen und Tests verhindern, daß eine negative Zahl bei einem rekursiven Aufruf als Parameter übergeben wird. Wir erhalten eine adäquate und effizientere Realisierung der Fakultätsfunktion, wenn wir die erforderliche Bereichsprüfung auslagern:

Version (4)

```
(defun fakultaet4 (zahl)
  (if (< zahl 0)
```

```
    "Fakultaet: unzulaessiges Argument"
    (berechne-fakultaet zahl)))
;; BERECHNE-FAKULTAET ist mit FAKULTAET1 identisch
(defun berechne-fakultaet (zahl)
  (if (= zahl 0)
      1
      (* zahl (berechne-fakultaet (- zahl 1)))))
```

Eine weitere Aufgabenstellung, für die sich eine rekursive Lösung anbietet, ist die Berechnung von *Fibonacci*-Zahlen:

Beispiel (5-2)
Die Folge der *Fibonacci*-Zahlen besteht aus den Zahlen für die gilt, daß jede der Zahlen die Summe der beiden ihr vorangehenden Zahlen ist:

0, 1, 1, 2, 3, 5, 8, 13, ... etc.

Ihr Bildungsprinzip läßt sich durch folgende rekursive Definition beschreiben:

$fib(n) = n$	gdw. n = 0 oder n = 1 und
$fib(n) = fib(n - 2) + fib(n - 1)$	gdw. n > 1.

Der entscheidende Unterschied gegenüber der Definition der Fakultätsfunktion liegt darin, daß es in dieser Definition eine Klausel gibt, die zwei rekursive Aufrufe von *fib* enthält. Funktionen dieses Typs werden als *bi-rekursive* bzw. *mehrfach rekursive* Funktionen bezeichnet. Auch diese Definition läßt sich sehr einfach als LISP-Funktion realisieren:

```
(defun fibonacci (zahl)
  (if (< zahl 0)
      "Fibonacci: unzulaessiges Argument"
      (berechne-fibonacci zahl)))
(defun berechne-fibonacci (zahl)
  (if (< zahl 2)
      zahl
      (+ (berechne-fibonacci (- zahl 2))
         (berechne-fibonacci (- zahl 1)))))
```

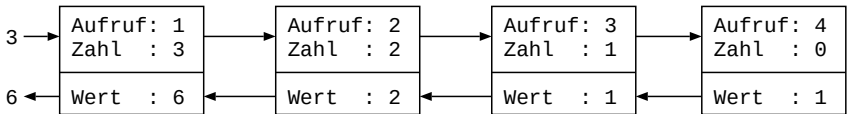
5.1.2 Ausfhrung rekursiver Funktionen

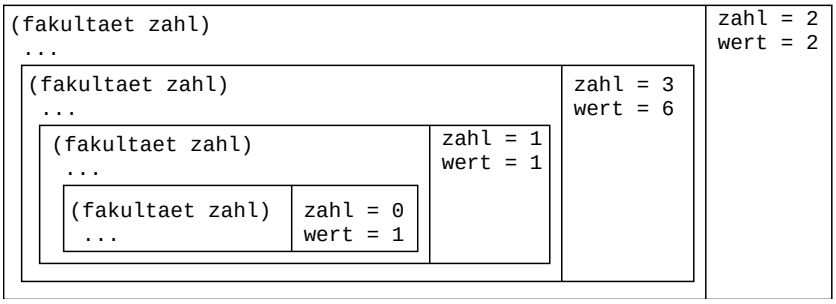
Die folgenden Beispiele illustrieren die Form des Berechnungsprozesses innerhalb der zuvor definierten rekursiven Funktionen:

Beispiel (5-3)

(A) Berechnung der Fakultät:

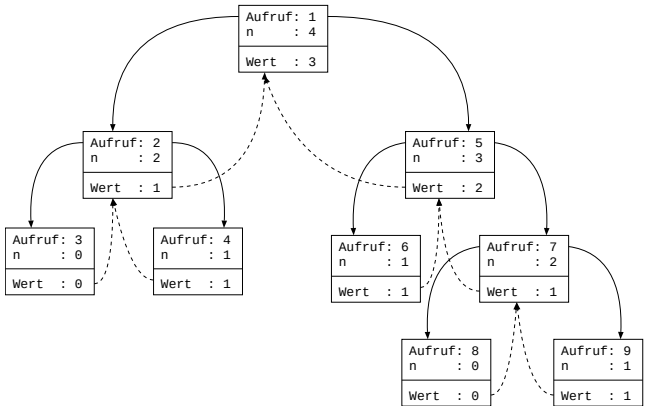
$$\begin{aligned} f(3) &= 3 * f(2) \\ &= 3 * 2 * f(1) \\ &= 3 * 2 * 1 * f(0) \\ &= 3 * 2 * 1 * 1 \\ &= 6 \end{aligned}$$





(B) Berechnung von Fibonacci-Zahlen

$$\begin{aligned} \mathit{fib}(4) &= \mathit{fib}(2) + \mathit{fib}(3) \\ &= \mathit{fib}(0) + \mathit{fib}(1) + \mathit{fib}(3) \\ &= 0 + \mathit{fib}(1) + \mathit{fib}(3) \\ &= 0 + 1 + \mathit{fib}(3) \\ &= 1 + \mathit{fib}(1) + \mathit{fib}(2) \\ &= 1 + 1 + \mathit{fib}(2) \\ &= 2 + \mathit{fib}(0) + \mathit{fib}(1) \\ &= 2 + 0 + 1 \\ &= 3 \end{aligned}$$



5.2 Rekursive Operationen auf Listen

Bei den bislang betrachteten Beispielen handelte es sich um *numerische Probleme*. Für uns interessanter sind nicht-numerische Aufgabenstellungen. Als ein Beispiel für solche Aufgabenstellungen werden wir jetzt verschiedene, auf Listen operierende Funktionen betrachten.

Listen[1] bilden eine rekursive Datenstruktur, deren Bildungsprinzip sich am natürlichsten durch eine rekursive Funktion beschreiben läßt:

```
(defun ab-&-aufbau (liste)
  (if (endp liste)
      ()
      (cons (first liste) (ab-&-aufbau (rest liste)))))
```

Diese Funktion kann als *Identitätsfunktion* für Listen aufgefaßt werden; denn sie bildet die als Argument übergebenen Listen auf sich selbst ab. Es ist ein heuristischer Aspekt, der sie für uns interessant macht: Die Funktion demonstriert, wie eine gegebene Liste zerlegt und durch CONS-Operationen aus elementareren Objekten wieder aufgebaut werden kann. Außerdem bildet diese Funktion den Nukleus für verschiedene Typen von Listenfunktionen, die wir im folgenden einführen werden.

Beispiel (5-4)

```
(AB-&-AUFBAU '(A B C)) ⟹
  (CONS A (AB-&-AUFBAU (B C)))
          (CONS B (AB-&-AUFBAU (C)))
                  (CONS C (AB-&-AUFBAU ()))
                          NIL
```

[1] Genaugenommen müßte man von CONSes und nicht von *Listen* sprechen.

(CONS A (CONS B (CONS C NIL))) $\Longrightarrow$ (A B C)

Der erste Funktionstyp, den wir betrachten werden, ist dadurch charakterisiert, daß auf alle Elemente der als Parameter übergebenen Liste eine Operation ausgeführt wird und als Wert die Liste der durch diese Operation erzeugten Objekte zurückgegeben wird:

Schema I

```
(defun Funktion (liste)
  (if (endp liste)
      ()
      (cons (Operation (first liste))
            (Funktion (rest liste)))))
```

Beispiel (5-5)

```
;;; Die Funktion ADD-THREE erwartet eine Liste von Zahlen als
;;; Argument und addiert 3 zu jeder Zahl.
(defun add-three (liste)
  (if (endp liste)
      ()
      (cons (+ 3 (first liste))
            (add-three (rest liste)))))
```

Dieses Schema läßt sich generalisieren, indem man zuläßt, daß durch sukzessive Anwendung einer möglicherweise komplexen Operation auf die Elemente der Argumentliste ein beliebiges Objekt (also nicht unbedingt eine *Liste*) erzeugt und als Funktionswert zurückgegeben wird:

Schema I'

```
(defun Funktion (liste)
  (if (endp liste)
      Wert
      (Operation (Auswerten(first liste))
                 (Funktion (rest liste)))))
```

Beispiel (5-6)

```
;;; Die Funktion OUR-PLUS addiert alle in der Liste enthaltenen Zahlen.
(defun our-plus (liste)
  (if (endp liste)
```

```
        0
        (+ (first liste) (our-plus (rest liste)))))
```

Ein anderer Typ rekursiver Funktionen unterscheidet sich von dem zuerst genannten darin, daß nicht auf alle Elemente der Liste eine Operation angewendet wird, sondern nur auf die Objekte, die einer best. Bedingung genügen:

Schema II

```
(defun Funktion (liste)
  (cond ((endp liste) Wert)
        (Bedingung
         (Operation (first liste)
                    (Funktion  (rest liste))))
        (t (Funktion (rest liste)))))
```

Funktionen dieses Typs können als **Filter** aufgefaßt werden. Ein gutes Beispiel für Funktionen dieser Klasse ist die *build-in* Funktion REMOVE (s.u.), die ein Objekt und eine Liste als Argumente nimmt und als Wert eine Kopie der Liste liefert, in der das Objekt nicht mehr vorkommt. Sie läßt sich leicht durch eine selbstdefinierte Funktion abbilden:

Beispiel (5-7)

```
;;; FUNKTION      :  OUR-REMOVE
;;; ARGUMENT(e)  :  Ein Objekt O und eine Liste L.
;;; WERT         :  Eine Liste L', die man aus L erhält, indem je-
;;;                 des Vorkommen von O entfernt wird. Allerdings
;;;                 werden nur atomare Objekte aus der Argument-
;;;                 liste entfernt.
(defun our-remove (objekt liste)
  (cond ((endp liste) ())
        ((not (eql (first liste) objekt))
         (cons (first liste)
               (our-remove objekt (rest liste))))
        (t (our-remove objekt (rest liste)))))
```

Es sollte nicht überraschen, daß sich natürlich nicht alle rekursiven Funktionen, die auf Listen operieren, auf eines der von uns formulierten Schemata reduzieren lassen[2].

[2]Weitergehende Versuche, Typen rekursiver Funktionen durch Angabe von ihnen zugrundeliegenden Schemata zu klassifizieren, finden sich u. a. in: E.Soloway '*From Problems to Programs Via Plans: The Content and Structure of Knowledge for Introductory LISP Programming*.' J.Educational Computing Research, Vol.1(2), 1985, S.157-172. D.Vorberg & R.Göbel '*Rekursionsschemata als Problemlösepläne*.' unveröff. Aufsatz, Fachbereich Psychologie, Uni.Marburg.

Zu den in Common LISP verfügbaren *Filter*-Funktionen gehören neben REMOVE die Funktionen REMOVE-IF und REMOVE-IF-NOT, die alle Objekte aus einer Liste entfernen, die die spezifizierte Bedingung erfüllen bzw. nicht erfüllen[3]:

REMOVE	*Objekt* *Liste*	[Funktion]
REMOVE liefert als Wert eine Kopie von *Liste*, in der alle Vorkommen von *Objekt* entfernt wurden.		

REMOVE-IF	*Test* *Liste*	[Funktion]
REMOVE-IF liefert als Wert eine Kopie von *Liste*, in der alle Objekte nicht mehr enthalten sind, die *Test* erfüllen.		

REMOVE-IF-NOT	*Test* *Liste*	[Funktion]
REMOVE-IF-NOT liefert als Wert eine Kopie von *Liste*, in der nur noch die Objekte enthalten sind, die *Test* erfüllen.		

Beispiel (5-8)

```
                (remove 'a '(a b a c a))   ⟹   (B C)
              (remove 'a '((a) b a c a))   ⟹   ((A) B C)
      (remove-if #'symbolp '(a 1 b 2 c 3))   ⟹   (1 2 3)
(remove-if-not #'listp '(() a (b c) 77 ((d) e)))   ⟹   (NIL (B C) ((D) E))
```

Abschließend noch ein Beispiel für eine bi-rekursive Funktion, die auf Listen operiert: Die Funktion ZAEHLE-ATOME liefert als Wert die Zahl **aller** Atome, die in einer Liste an beliebiger Position enthalten sind.

Beispiel (5-9)

```
(defun zaehle-atome (liste)
  (cond ((null liste) 0)
        ((atom liste) 1)
        (t (+ (zaehle-atome (first liste))
              (zaehle-atome (rest liste))))))
```

[3]Genau wie bei MEMBER kann auch bei REMOVE durch das *Test-Keyword* die zu verwendende Vergleichsfunktion (*Default*: EQ) spezifiziert werden.

5.3 Endrekursive Funktionen

Charakteristisch für *endrekursive Funktionen* ist, daß mit der Terminierung eines Funktionsaufrufs der Berechnungsprozeß abgeschlossen ist; d.h. der Wert wird zurückgegeben, ohne daß weitere Operationen mit ihm ausgeführt werden. Die meisten der im letzten Abschnitt definierten Listenfunktionen z.B. verwenden den bei der Terminierung eines Funktionsaufrufs gelieferten Wert (häufig: NIL), zur Generierung der Resultatliste (typischerweise: CONS).

Endrekursive Funktionen können sehr effizient verarbeitet werden: Wenn mit der Terminierung der Funktion der Berechnungsprozeß abgeschlossen ist, dann müssen zwischen dem ersten Aufruf der Funktion und ihrer Terminierung keine Werte zwischengespeichert werden; d.h. die sonst bei rekursiven Funktionen aufwendigen Stackoperationen können auf ein Minimum reduziert werden. Einige LISP-Implementierungen bieten *optimierende* Compiler, die endrekursive Funktionen vor dem eigentlichen Compilieren zunächst in iterative Funktionen umwandeln und so sehr effiziente Resultate garantieren.

Bevor wir einige Beispiele für endrekursive Funktionen entwickeln, präzisieren wir den Begriff der endrekursiven Funktion:

> Wenn ein Problem P_1 in ein anderes Problem P_2 überführt werden kann, so daß nach der Lösung von P_2 keine weiteren Berechnungen mehr notwendig sind, dann wird P_2 als *Reduktion* von P_1 bezeichnet.
>
> Eine rekursive Funktion, in der jeder rekursive Aufruf eine Reduktion darstellt, wird als *endrekursive* (*tail recursive*) Funktion bezeichnet.

Beispiel (5-10)

```
(defun my-reverse (liste)
  (if (endp liste)                          ; Abbruchbedingung.
      ()
      (append (my-reverse (rest liste))     ; Rekursiver Aufruf.
              (cons (first liste) ()))))
```

MY-REVERSE ist nicht endrekursiv: Der Wert, den die Funktion liefert, wenn ihr Argument eine nicht-leere Liste ist, wird gebildet, indem der Wert des rekursiven Aufrufs mit einer das erste Element der Liste enthaltenden Liste verknüpft wird:

1	(my-reverse '(a b))	Aufruf
2	(append (my-reverse (rest '(a b))) (cons (first '(a b)) ()))	Rekursiver Abstieg
3	(append (my-reverse '(b) (cons 'a ()))	
4	(append (my-reverse '(b) (a))	
5	(append (append (my-reverse (rest '(b)))	
	(cons (first '(b)) ())) '(a))	
6	(append (append (my-reverse ()) (cons 'b ())) '(a))	
7	(append (append (my-reverse ()) '(b)) '(a))	
8	(append (append () '(b)) '(a))	Terminierung
9	(append '(b) '(a))	Nachklappern
10	(b a)	

Um eine endrekursive Fassung der Funktion zu erhalten, muß die Resultatsliste bereits beim rekursiven Abstieg aufgebaut werden, so daß das *Nachklappern* entfällt. Zu diesem Zweck wird ein *Akkumulator* verwendet, in dem die beim rekursiven Abstieg gewonnenen Werte gesammelt werden, und der nach der Terminierung des Funktionsaufrufs zurückgegeben wird, ohne daß weitere Operationen auf ihm ausgeführt werden.

Beispiel (5-11)
Eine endrekursive Variante der Funktion MY-REVERSE sollte sich wie folgt verhalten:

1.Aufruf	Parameter	Wert
1. Aufruf	(A B C D)	()
2. Aufruf	(B C D)	(A)
3. Aufruf	(C D)	(B A)
4. Aufruf	(D)	(C B A)
5. Aufruf	()	(D C B A)

```
;;; FUNKTION   :  REVERSE+
;;; KOMMENTAR  Endrekursive Fassung der Funktion MY-REVERSE
  (defun reverse+ (liste)
    (reverse++ liste NIL))          ; Der Akkumulator wird mit NIL initialisiert.
  (defun reverse++ (liste akku)
    (if (endp liste)
        akku
        (reverse++ (rest liste) (cons (first liste) akku))))
        ; Aktualisierung des Akkumulators beim rekursiven Abstieg.
```

Beispiel (5-12)

```
;;; FUNKTION       :  COUNT-ATOMS
;;; ARGUMENT(e)    :  Eine beliebige Liste.
;;; Wert           :  Eine Zahl, die angibt, wieviele Atome die Liste enthält.
;;; (a) nicht endrekursive Fassung:
(defun count-atoms (liste)
  (cond ((null liste) 0)
        ((atom liste) 1)
        (t (+ (count-atoms (first liste))
              (count-atoms (rest liste))))))
;;; (b) endrekursive Fassung:
(defun count-atoms+ (liste)
  (counts liste 0))             ; Der Akkumulator wird mit 0 initialisiert.
(defun counts (liste akku)
  (cond ((null liste) akku)
        ((atom liste) (+ akku 1))
```

```
(t (counts (first liste) (counts (rest liste) akku)))))
```

Trace der endrekursiven Fassung:

```
(counts '((a b) (c)) 0)
(counts (a b) (counts ((c)) 0))
(counts (a b) (counts (c) (counts () 0)))
(counts (a b) (counts (c) 0))
(counts (a b) (counts c (counts () 0)))
(counts (a b) (counts c 0))
(counts (a b) (+ 0 1))
(counts (a b) 1)
(counts a (counts (b) 1))
(counts a (counts b (counts () 1)))
(counts a (counts b (+ 0 1))
(counts a (counts b 1))
(counts a (+ 1 1))
(counts a 2)
(+ 2 1)
3
```

5.4 Mengen

Zum Abschluß dieses Kapitels demonstrieren wir an einem allen vertrauten Gegenstandsbereich die Leistungsfähigkeit rekursiver Funktionen. Wir werden eine Reihe von Funktionen entwickeln, die die wichtigsten mengentheoretischen Relationen abbilden. Dabei beschränken wir uns zunächst auf Mengen ersten Ordnung; d.h. auf Mengen, die nur einfache Objekte (und keine Listen) als Elemente enthalten.

Wir werden Mengen durch Listen repräsentieren. Zwei Punkte gilt es zu beachten:

1. Vergleicht man zwei Mengen, ist anders als bei Listen die Reihenfolge ihrer Elemente irrelevant. So gilt {a, b} = {b, a}; (a b) und (b a) dagegen sind zwei nicht–identische Listen.

2. In Mengen kommt jedes Element nur einmal vor: {a} ∪ {a, b} = {a, b}; in Listen können Objekte mehrfach vorkommen.

Aus diesen Gründen definieren wir zunächst eine Funktion MAKE-SET, die sicherstellt, daß in den zur Repräsentation von Mengen verwendeten Listen kein Objekt mehrfach vorkommt:

```
;;; FUNKTION      :  MAKE-SET
;;; ARGUMENT(e)   :  Eine beliebige Liste L, die keine weiteren Listen
;;;                  enthält.
;;; WERT          :  Eine Liste, in der jedes Element aus L nur ein-
;;;                  mal vorkommt; d.h., diese Funktion konvertiert
;;;                  beliebige Listen in Mengen.

(defun make-set (liste)
  (cond ((endp liste) nil)
        ((is-element-of-p (first liste) (rest liste))
         (make-set (rest liste)))
        (t (cons (first liste) (make-set (rest liste))))))
```

Listen, die einfache Mengen repräsentieren, erfüllen die folgenden beiden Prädikate:

```
;;; FUNKTION      :  SET-P
;;; ARGUMENT(e)   :  Eine Liste.
;;; WERT          :  T gdw. es kein Objekt gibt, das in der Liste mehr
;;;                  als einmal vorkommt; sonst NIL.

(defun set-p (liste)
  (cond ((endp liste) t)
        ((is-element-of-p (first liste) (rest liste)) nil)
        (t (set-p (rest liste)))))
```

```
;;; FUNKTION      :  SIMPLE-SET-P
;;; ARGUMENT(e)   :  Eine Menge.
;;; WERT          :  T gdw. die Menge nur atomare Objekte (exklu-
;;;                  sive der leeren Liste) enthält; sonst NIL.

(defun simple-set-p (set)
  (and (set-p set)
       (= (length set)
          (length (remove-if #'listp set)))))
```

```
;;; FUNKTION      :  CARDINALITY
;;; ARGUMENT(e)   :  Eine Menge.
;;; WERT          :  Die Zahl der Elemente dieser Menge.

(defun cardinality (set)
  (length set))
```

5.4.1 Operationen auf Mengen

```
;;; FUNKTION      :  UNION*
;;; ARGUMENT(e)  :  Zwei Mengen SET1 und SET2.
;;; WERT          :  Die Menge aller Elemente, die in SET1 oder
;;;                  SET2 vorkommen.
;;;                  Die Funktion erzeugt die Vereinigungsmenge
;;;                  von SET1 und SET2.
(defun union*[4](set1 set2)
  (cond ((endp set1) set2)
        ((is-element-of-p (first set1) set2)
         (union* (rest set1) set2))
        (t (cons (first set1)
           (union* (rest set1) set2)))))
```

```
;;; FUNKTION      :  INTERSECTION*
;;; ARGUMENT(e)  :  Zwei Mengen SET1 und SET2.
;;; WERT          :  Die Menge aller Elemente, die in SET1 und
;;;                  SET2 enthalten sind. Die Funktion berechnet
;;;                  die Schnittmenge von SET1 und SET2.
(defun intersection* (set1 set2)
  (cond ((endp set1) ())
        ((is-element-of-p (first set1) set2)
         (cons (first set1) (intersection* (rest set1) set2)))
        (t (intersection* (rest set1) set2))))
```

```
;;; FUNKTION      :  DIFFERENCE*
;;; ARGUMENT(e)  :  Zwei Mengen SET1 und SET2.
;;; WERT          :  Die Menge aller Elemente aus SET1, die nicht
;;;                  in SET2 vorkommen. Die Funktion erzeugt die
;;;                  Differenzmenge von SET1 und SET2.
(defun difference* (set1 set2)
  (cond ((endp set1) ())
        ((is-element-of-p (first set1) set2)
         (difference* (rest set1) set2))
        (t (cons (first set1) (difference* (rest set1) set2)))))
```

;;; Die Funktion DIFFERENCE* kann verwendet werden, um INTERSECTION* zu definieren:

```
(defun intersection** (set1 set2)
  (difference* set1 (difference* set1 set2)))
```

[4]Eine einfache, konzeptuell aber unbefriedigende Möglichkeit, die Vereinigung zweier Mengen zu realisieren, besteht darin, zunächst mit APPEND beide Listen zu verschmelzen und anschließend die mehrfach

5.4.2 Mengentheoretische Prädikate

```
;;; FUNKTION      :  IS-ELEMENT-OF-P
;;; ARGUMENT(e)   :  Ein atomares Objekt und eine Menge.
;;; WERT          :  T gdw. das Objekt in der Menge enthalten ist;
;;;                  sonst NIL. Das Prädikat realisiert die Element-
;;;                  beziehung.
```

```
(defun is-element-of-p (object set)
  (if (member object set)
      t
      nil))
```

```
;;; FUNKTION      :  SUBSET-P
;;; ARGUMENT(e)   :  Zwei Mengen SET1 und SET2.
;;; WERT          :  T gdw. alle Elemente von SET1 auch in SET2
;;;                  vorkommen; sonst NIL. Das Prädikat prüft, ob
;;;                  SET1 eine Teilmenge von SET2 ist.
```

```
(defun subset-p (set1 set2)
  (cond ((endp set1) t)
        ((is-element-of-p (first set1) set2)
         (subset-p (rest set1) set2))
        (t nil)))
```

```
;;; FUNKTION      :  REAL-SUBSET-P
;;; ARGUMENT(e)   :  Zwei Mengen SET1 und SET2.
;;; WERT          :  T gdw. SET1 eine Teilmenge von SET2 ist und
;;;                  SET2 mindestens ein Element enthält, das nicht
;;;                  in SET1 enthalten ist; sonst NIL.
;;;                  Das Prädikat prüft, ob SET1 eine *echte* Teil-
;;;                  menge von SET2 ist.
```

```
(defun real-subset-p (set1 set2)
  (and (> (length set2) (length set1))
       (subset-p set1 set2)))
```

```
;;; FUNKTION      :  IDENTICAL-SETS-P
;;; ARGUMENT(e)   :  Zwei Mengen.
;;; WERT          :  T gdw. beide Mengen dieselben Element enthal-
;;;                  ten, d.h. sie identisch sind.
```

---

vorkommenden Elemente zu entfernen:
(defun union (set1 set2) (make-set (append set1 set2))).

```
(defun identical-sets-p (set1 set2)
  (and (subset-p set1 set2)
       (subset-p set2 set1)))
```

Kapitel 6

Iteration

Nachdem wir uns im letzten Kapitel mit den Grundzügen rekursiver Programmierung auseinandergesetzt haben, werden wir uns in diesem Kapitel mit den Kontrollstrukturen vertraut machen, die Common-LISP für die Entwicklung iterativer Prozeduren bereitstellt. Ähnlich wie die allgemeinen Kontrollstrukturen (COND, IF, WHEN, UNLESS, CASE, ...) in Kapitel 4 werden sie nicht durch Funktionen, sondern durch Makros realisiert.

6.1 Begrenzte Iteration

Die ersten beiden iterativen Kontrollstrukturen DOTIMES und DOLIST ermöglichen es, eine Sequenz von Operationen n-mal bzw. einmal für jedes Element einer gegebenen Liste auszuführen.

6.1.1 FOR I = ... TO ... DO

Der DOTIMES-Makro erlaubt die Formulierung von *FOR*-Schleifen, wie man sie aus prozeduralen Sprachen wie z.B. PASCAL kennt. Die Syntax ist - wie die der meisten in diesem Kapitel vorgestellten Makros - gewöhnungsbedürftig. Auf das DOTIMES-Schlüsselwort folgt eine Liste, die die folgende Angaben enthält:

1. Das erste Element der Liste ist eine Variable (*Zählvariable*).

2. Es folgt eine Form, die festlegt, wie oft die Formen des folgenden Anweisungsblocks zu evaluieren sind; d.h. diese Form muß zu einem Integer evaluieren.

3. Optional kann noch eine Form angegeben werden, die nach dem letzten Schleifendurchgang evaluiert wird und deren Wert als Wert der DOTIMES-Form zurückgegeben wird.

DOTIMES	(*Var Zähl-Form* [*Resultat*]) {*Form*}*	[Makro]
Bei der Evaluierung einer DOTIMES-Form wird zunächst *Zähl-Form* evaluiert. Diese Form muß zu einer Zahl n evaluieren. Anschließend wird der Anweisungsblock n-mal ausgeführt, wobei die Variable *Var* nacheinander an die Werte 0 bis n-1 gebunden ist. Wenn n $\leq$ 0, dann wird der Anweisungsblock nicht ausgeführt. Wenn die *Resultat*-Form weggelassen wird, evaluiert die DOTIMES–Form zu NIL.		

Zur Illustration greifen wir auf die Fakultäts- und Fibonacci-Funktion aus dem letzten Kapitel zurück:

Beispiel (6-1)

```
;; rekursive Version:
(defun fakultaet (n)
  (if (<= n 0)
      1
      (* n (fakultaet (- n 1)))))
;; iterative Version:
(defun fakultaet (n)
  (let ((resultat 1))
    ;; Generierung und Initialisierung einer Variablen zur Berechnung
    ;; des Resultats
    (dotimes (x n resultat)
      (setq resultat (* resultat (+ x 1))))))
      ;; Aktualisierung der Resultatsvariable:
      ;; Zur Berechnung des korrekten Ergebnisses ist es notwendig,
      ;; den Wert von x um 1 zu erhöhen, da x Werte zwischen 0 und
      ;; n-1 annimmt.
```

Beispiel (6-2)

```
;; rekursive Version:
(defun fibonacci (n)
  (if (<= n 2)
      1
      (+ (fibonacci (- n 2)) (fibonacci (- n 1)))))
;; iterative Version:
(defun fibonacci (n)
```

```
(let ((vorletzte 0) (letzte 1) (x 0))
  ;; Drei Variablen sind erforderlich:
  ;; Sie speichern die letzte und vorletzte gefundene Fibonacci-Zahl.
  (dotimes (y (- n 1) letzte)          ; Komplexe Aktualisierung:
     (setq x (+ letzte vorletzte)      ; Neue Fibonacci-Zahl berechnet.
           vorletzte letzte            ; Aktualisierung der Variablen.
           letzte x))))
```

Spätestens die Fibonacci-Funktion demonstriert, daß es iterativen Definitonen gegenüber rekursiven Definitionen häufig an Transparenz mangelt: Während die rekursive Version ein unmittelbar einsichtiges Berechnungsprinzip erkennen läßt, erschwert in der iterativen Version allein schon die Zahl der zusätzlichen Variablen ein Verständnis dieser Definition. Ein weiterer Unterschied zwischen der iterativen und der rekursiven Realisierung der Fibonacci-Funktion besteht darin, daß die iterative Funktion Fibonacci-Zahlen durch Vorwärtsverkettung berechnet:

vorletzte		*letzte*		x
0	+	1	$\Rightarrow$	1
1	+	1	$\Rightarrow$	2
1	+	2	$\Rightarrow$	3
2	+	3	$\Rightarrow$	5
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Aber auch die Effizienzvorteile gegenüber rekursiven Definitionen werden deutlich: Während bei jeden rekursiven Aufruf einer Funktion ein Satz neuer Parameter erzeugt und verwaltet werden muß, ist bei einer iterativen Funktion der Speicherbedarf konstant. Es werden für die gesamte Berechnung nur die zu Beginn explizit generierten Variablen benötigt.

6.1.2 Iteration über Listen

DOLIST erlaubt es, für jedes Element einer Liste eine Folge von Anweisungen auszuführen. Die Syntax des DOLIST-Makros ist mit der des DOTIMES-Makros identisch. Der einzige semantische Unterschied liegt darin, daß die auf die Schleifenvariable folgende Form zu einer Liste evaluieren muß, an deren Elemente die Zählvariable nacheinander gebunden wird. Iterative Versionen der Systemfunktionen REVERSE und LENGTH verdeutlichen die Einsatzmöglichkeiten von DOLIST:

DOLIST	(*Var List-Form* [*Resultat*])	{*Form*}*	[Makro]

Bei der Evaluierung einer DOLIST-Form wird zunächst *List-Form* evaluiert. Diese Form muß zu einer Liste evaluieren. Alle Elemente dieser Liste werden anschließend nacheinander, von links nach rechts vorgehend, an die Variable *Var* gebunden, und der Anweisungsblock wird unter Berücksichtigung dieser Bindung ausgeführt. Wenn die *Resultat*-Form weggelassen wird, evaluiert die DOLIST-Form zu NIL.

Beispiel (6-3)

```
;; MY-REVERSE
(defun my-reverse (liste)
  (let ((resultat ()))
    (dolist (var liste resultat)
      (setq resultat (cons var resultat)))))

;; MY-LENGTH
(defun my-length (liste)
  (let ((zahl 0))
    (dolist (x liste zahl)
      (setq zahl (+ zahl 1)))))
```

Wie die Beispiele in diesem Abschnitt zeigen, basieren Funktionsdefinitionen, deren zentrale Kontrollstruktur der DOTIMES bzw. DOLIST-Makro ist, auf folgendem Schema:

```
(defun Funktion (...)
  (GENERIERE Resultatsvariable [Hilfsvariablen])
    (DO{TIMES|LIST} (Zählvariable Wertform Resultatsform)
      ......
      (AKTUALISIERE Resultatsvariable [Hilfsvariablen])
      ......      ))
```

6.2 Iteration ohne Grenzen

Die beiden leistungsfähigsten, aber auch komplexesten Makros zur Realisierung iterativer Funktionsdefinitionen sind die Makros DO und DO*:

DO	({(*Variable* [*Anfangswert* [*Neuer-Wert*]])}*) (*End-Test* {*Resultat*}*) {*Form*}*	[Makro]
Zunächst werden die spezifizierten Variablen generiert und initialisiert. Bei jedem weiteren Durchgang werden sie auf Grundlage der *Neuer-Wert* Formen aktualisiert. Vor jedem Schleifendurchgang wird überprüft, ob die Abbruchbedingung *End-Test* erfüllt ist. Tritt diese Situation ein, werden die *Resultat*-Formen evaluiert, und der Wert der letzten Form wird als Wert der DO-Form zurückgegeben. Anderenfalls werden die Formen des Schleifenkörpers evaluiert.		

Eine genauere Vorstellung von der syntaktischen Struktur einer DO-Form vermittelt die folgende Darstellung:

(DO ((*Variable*$_1$ *Anfangswert*$_1$ *Neuer-Wert*$_1$) (1) Variableninitialisierung
(*Variable*$_2$ *Anfangswert*$_2$ *Neuer-Wert*$_2$) und -aktualisierung
⋮ ⋮ ⋮
⋮ ⋮ ⋮
(*Variable*$_n$ *Anfangswert*$_n$ *Neuer-Wert*$_n$))
(*Test* . *Resultat*) (2) Abbruchbedingung
Form$_1$ (3) Anweisungsblock
Form$_2$
⋮
Form$_m$)

- *Syntax*:
  Jede Variablendeklaration besteht aus einem Variablennamen, dem Anfangswert der Variablen und einer Update-Form *Neuer-Wert*. Die letzten beiden Angaben sind optional. Die Abbruchbedingung besteht aus einer *Test*-Form und einer möglicherweise leeren Folge von *Resultat*-Formen. Der Anweisungsblock besteht aus einer Folge von Formen.

- *Semantik*:

  – *Initialisierung/Aktualisierung der Variabeln*: Zu Beginn werden die Variablen in (1) an die spezifizierten Anfangswerte gebunden. Variablen, für die kein An-

fangswert spezifiziert wurde, erhalten den Wert NIL. Die Wertzuweisung erfolgt **parallel**. Vor dem nächsten Durchgang werden alle Variablen, für die eine Update-Form angegeben wurde, an den Wert dieser Form gebunden.

- *Überprüfung der Abbruchbedingung*: Ist die Abbruchbedingung *Test* in (2) erfüllt, werden die Resultat-Formen evaluiert, und der Wert der letzten Form wird als Wert der DO-Form zurückgegeben.
- *Ausführung der Anweisungen*: Anschließend werden alle Formen des Anweisungsblocks (3) evaluiert. Dieser Vorgang wiederholt sich solange, bis die Abbruchbedingung erfüllt ist.

Beispiel (6-4)

```
;;; COUNT-ATOMS
;;; Zählt die Atome einer beliebig komplexen Liste.

;; iterativ-rekursive Version:
(defun count-atoms (liste)
  (let ((zahl 0))
    (dolist (x liste zahl)
      (if (not (listp x))
          (setq zahl (+ zahl 1))
          (setq zahl (+ zahl (count-atoms x)))))))

;; iterative Version:
(defun count-atoms (liste)
  (do ((reste-liste liste (rest reste-liste))
       (resultat 0))
      ((endp reste-liste) resultat)
      (unless (null (first reste-liste))
        (if (listp (first reste-liste))
            ;; Erstes Element von RESTE-LISTE ist eine Liste:
            (setq reste-liste
                  (append reste-liste
                          (list (first (first reste-liste))
                                (rest (first reste-liste)))))
            ;; Erstes Element von RESTE-LISTE ist ein Atom:
            (setq resultat (+ resultat 1))))))
```

Wie unterscheiden sich die beiden Versionen von COUNT-ATOMS? Im ersten Fall werden die Elemente der Liste "direkt" ausgewertet: ist es ein Atom, wird die Resultatsvariable ZAHL aktualisiert; ist es eine Liste, wird die Funktion rekursiv mit dieser Liste aufgerufen. In der zweiten Version werden Sublisten schrittweise aufgelöst.

DO*	({(*Variable* [*Int* [*Neu*]])}*) (*End-Test* {*Resultat*}*) {*Form*}*	[Makro]

Eine DO*-Form unterscheidet sich von einer DO-Form allein dadurch, daß die Wertzuweisung bei der Initialisierung und Aktualisierung der Variablen sequentiell durchgeführt wird.

Der Unterschied zwischen DO und DO* entspricht dem zwischen LET und LET*: Erfolgt in einer DO-Form die Initialisierung bzw. Aktualisierung der Variablen *parallel*, so erfolgt sie in einer DO*-Form *sequentiell*; d.h. in einer DO*, nicht aber in einer DO-Form kann bei der Berechnung des Wertes einer Variablen auf die Werte der voranstehenden Variablen zugegriffen werden.

Beispiel (6-5)

```
;; MY-REVERSE mit DO
(defun my-reverse (liste)
  (do ((hliste liste (rest hliste))
       (resultat nil (cons (first hliste) resultat)))
      ((endp hliste) resultat)))
;; MY-REVERSE mit DO*
;; umständliche Reformulierung
(defun my-reverse (liste)
  (do* ((hliste liste (rest hliste))
        (resultat (unless (endp liste) (list (first liste)))
                  (cons (first hliste) resultat)))
       ((endp (rest hliste)) resultat)))
```

DOTIMES und DOLIST-Formen lassen sich ohne Probleme in äquivalente DO bzw. DO*Formen überführen:

```
(dotimes (Var Zahl Resultat) ...)   ≡   (do ((Var 0 (+ Var 1)))
                                            ((= Var Zahl) Resultat)
                                            ...)
```

(dolist (Var Liste Resultat) ...) ≡

```
(do* ((l Liste (rest l))
      (element (first l) (first l)))
     ((endp l) Resultat)
     ...)
```

Während DOTIMES- und DOLIST-Formen zur Generierung einer Resultatsvariablen in eine LET/LET*-Form eingebettet werden müssen, kann die Resultatsvariable in DO/DO*-Formen direkt innerhalb des Variablendeklarationsteils erzeugt werden.

6.3 Schleifen ohne Ende

Als letzte iterative Kontrollstruktur betracheten wir das LOOP-Makro: Es erlaubt mit einfachsten Mitteln Schleifen zu erzeugen, die an beliebiger Position (oder auch nie) verlassen werden können.

LOOP	{*Form*}*	[Makro]
Die Formen werden solange immer wieder ausgeführt, bis durch eine RETURN–Form ihre Ausführung explizit terminiert wird.		

Beispiel (6-6)

```
(defun endlosschleife ()
  ;; nicht-terminierende Schleife
  (loop
    "hier haeng ich nun und kann nicht mehr"))
```

Eine LOOP-Schleife kann an beliebiger Position durch Verwendung einer RETURN-Form verlassen werden.

RETURN	[*Resultat*]	[Makro]
Die Evaluierung einer RETURN-Form bewirkt, daß der implizit durch ein iteratives Konstrukt wie DO oder LOOP gebildetete Block verlassen wird. Ist eine Resultat-Form spezifiziert, dann wird der Wert dieser Form zurückgegeben; sonst NIL.		

Eine LOOP-Form kann beliebig viele RETURN-Formen enthalten.

Beispiel (6-7)

```
(defun dumme-addition (zahl)
  (let ((wert 0))
    (unless (= zahl 0)
      (loop
        (when (> wert 100)
          (return "obere Grenze ueberschritten"))
        (when (< wert -100)
          (return "untere Grenze ueberschritten"))
        (setq wert (+ wert zahl))))))
```

LOOP-Schleifen eignen sich ideal zur Entwicklung von *Metainterpretern*[1]:

Beispiel (6-8)

```
;; Sehr einfacher Meta-Interpreter
(defun interpreter ()
  ;; LISP in LISP in seiner 'reinsten' Form
  (loop
    (princ[2]"Mein Prompt :::::> ")   ; Ausgabe eines Prompts
    (print (eval (read)))))           ; Lese-Evaluiere-Drucke

;; Einfache DOS-Shell für Allegro Common LISP.
(defun my-interpreter ()
  (loop
    (print-prompt)
    (let ((eingabe (read)))
      (case eingabe
          (laden                      ; Laden von LISP-Programmen:
            (print-prompt)            ; Ausgabe eines Prompts.
            (print "Name der zu ladenden Datei: ")
            (load (read)))            ; Laden der Datei.
          (löschen                    ; Löschen von LISP-Programmen:
```

[1] Metainterpreter bilden einen bestimmten Typ von *Metaprogrammen*: Metaprogramme operieren auf anderen Programme; d.h. sie behandeln andere Programme als Daten. Interpreter und Compiler z.B. gehören zu dieser Klasse von Programmen. *Metainterpreter* sind Interpreter für eine Sprache, die in ihr selbst geschrieben sind. Sie ermöglichen es, auf das der Sprache zugrundeliegende Ausführungsmodell zuzugreifen und so mit einfachen Mitteln komplexe Aufgabenstellungen zu lösen

```
            (print-prompt)          ; Ausgabe eines Prompts.
            (print "Name der zu löschenden Datei: ")
            (delete-datei (read))   ; Löschen der Datei.
          (ende                     ; Verlassen des Metainterpreters
            (return 'bye))
          (t (print (eval eingabe)))))))

(defun print-prompt ()
  (terpri)
  (princ "? ")

;; Mögliche Nutzung des Metainterpreters:
> (my-interpreter)
? (= 7 93)
  100
? laden
  "Name der zu ladenden Datei: " "test1.lsp"
  FILE TEST1.LSP loaded
? löschen
  "Name der zu ladenden Datei: " "test1.lsp"
? ende
  BYE
```

[1] Eine genaue Beschreibung der Syntax und Semantik von Ausgabefunktionen wie PRINC, PRINT, ... etc. folgt in Kapitel 9.

Kapitel 7

Mapping

"Mapping" ist eine Form der Iteration, bei der eine Funktion nacheinander auf die Komponenten einer oder mehrerer Sequenzen angewendet wird. Der Typ **Sequenz** unfaßt Listen und eindimensionale Vektoren (und damit auch Strings). Bei den *Mapping*-Funktionen handelt es sich also um Funktionen der 2. Stufe: Sie wenden eine andere Funktion sukzessive auf eine Folge von Argumenten an, die in einer oder mehreren Listen zusammengefaßt sind. Die Zahl der Listen muß mit der Stelligkeit der angewandten Funktion übereinstimmen.

Die leistungsfähigste *Mapping*-Funktion heißt MAP:

1. Sie ermöglicht zu entscheiden, ob die Werte der einzelnen Funktionsaufrufe zu einem Wert akkumuliert werden oder nicht.

2. Der Typ des von der MAP-Form gelieferten Wertes (z.B. Liste oder String) kann festgelegt werden.

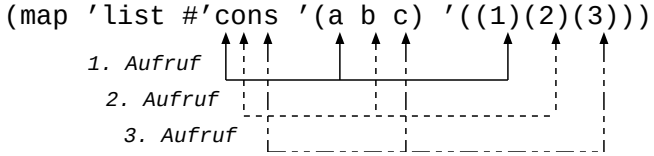
3. Anders als die anderen *Mapping*-Funktionen akzeptiert sie nicht nur Listen sondern auch andere Typen von Sequenzen als nicht-funktionale Argumente.

MAP *Ergebnistyp* *Funktion* *Sequenz*$^+$ [Funktion]
Die Zahl der angegebenen Sequenzen muß mit der Zahl der Argumente von *Funktion* übereinstimmen. *Funktion* wird jeweils mit den i-ten Elementen ($1 \leq i \leq$ Länge der kürzesten Sequenz) aller Sequenzen aufgerufen. Sind sie unterschiedlich lang, terminiert dieses Verfahren, sobald das Ende der kürzesten Sequenz erreicht ist. Der Wert der MAP-Form wird bestimmt durch die Ergebnisse der einzelnen Funktionsaufrufe und ist vom Typ *Ergebnistyp*. Wird als Ergebnistyp NIL angegeben, evaluiert die MAP-Form zu NIL.

Beispiel (7-1)

```
> (map 'list #'numberp '(1 NIL (2 3) -77.3))
  (T NIL NIL T)
> (map 'string #'(lambda (x) (if (numberp x) #\1 #\0)) '(7 autos () 2))
  "1001"
> (map 'list #'cons '(a b c) ' ((1) (2) (3)))
  ((a 1)(b 2)(c 3))
```

Die folgende Abbildung illustriert das Verarbeitungsprinzip der *Mapping*-Funktionen:



Die anderen Mapping-Funktionen, die wir jetzt einführen werden, lassen sich wie folgt klassifizieren:

	ignoriert Werte	*sammelt Werte*	*verknüpft Werte*
operiert auf den Elementen	MAPC	MAPCAR	MAPCAN
operiert auf Restlisten	MAPL	MAPLIST	MAPCON

7.1 Mapping auf Listenelementen

Die MAPCAR-Funktion wendet die als Argument übergegebene Funktion auf jedes Element der Argumentlisten an und liefert als Wert die Liste aller so berechneter Werte.

MAPCAR *Funktion Liste** [Funktion]

Die Funktion MAPCAR wendet *Funktion* nacheinander auf alle Elemente der Argument-Listen an und liefert als Wert die Liste aller erzielten Werte. Wenn die Listen nicht dieselbe Länge besitzen, terminiert die Iteration, nachdem die kürzeste Liste abgearbeitet ist.

MAPCAN *Funktion Liste** [Funktion]

Die Funktion MAPCAN unterscheidet sich von MAPCAR nur dadurch, daß die Ergebnisse der Funktion mit NCONC verknüpft werden.

NCONC ist das *destruktive* Gegenstück zur Funktion APPEND; d.h. es werden tatsächlich die als Argumente übergebenen Listen miteinander verknüpft und keine neue Liste erzeugt.

Beispiel (7-2)

```
> (setq *liste1* '(a) *liste2* '(b c))
  (B C)
> (append *liste1* *liste2*)
  (A B C)
> *liste1*
  (A)
> *liste2*
  (B C)
> (nconc *liste1* *liste2*)
  (A B C)
> *liste1*
  (A B C)
> *liste2*
  (B C)
>(setf (second *liste1*) 'neuer-wert)
  (A NEUER-WERT C)
> *liste1*
  (A NEUER-WERT C)
> *liste2*
  (NEUER-WERT C)
```

MAPCAN läßt sich auf MAPCAR zurückführen:

$$<\text{mapcan } f\ x_1 \ldots x_n> \equiv <\text{apply \#'nconc } <\text{mapcar } f\ x_1 \ldots x_n>>$$

Eine nicht-destruktive Variante kann man durch die Ersetzung von NCONC durch APPEND erhalten.

MAPC	*Funktion* *Liste**	[Funktion]
Die Funktion MAPC arbeitet wie MAPCAR, akkumuliert aber nicht die aus den Funktionsanwendungen resultierenden Werte. Eine MAPC-Form liefert als Wert ihre erste Argumentliste.		

Beispiel (7-3)

```
> (mapcar #'numberp '(2 "hallo" 99 (+ 3 4) karl))
  (T NIL T NIL NIL)
> (mapcar #'(lambda (x) (* x 10)) '(1 2 3 4 5))
  (10 20 30 40 50)
> (mapcar #'cons '(a b c) '((1) (2) (3)))
  ((A 1) (B 2) (C 3))
> (mapcan #'cons '(a b c) '((1) (2) (3)))
  (A 1 B 2 C 3)
> (mapc #'cons '(a b c) '((1) (2) (3)))
  (A B C)
> (mapc #'(lambda (x y) (print (cons x y))) '(a b c) '((1) (2) (3)))
  (A 1)      Seiteneffekt
  (B 2)      Seiteneffekt
  (C 3)      Seiteneffekt
  (A B C)    Wert

;; Noch einmal werden die Atome einer Liste gezählt:
;; (rekursiv/iterativ vgl. Formulierung mit DOLIST in Kap.6)
(defun count-atoms (liste)
  (cond ((null liste) 0)
        ((atom liste) 1)
        (t (apply #'+ (mapcar #'count-atoms liste)))))

;; Diese Funktion berechnet die Verschachtelungstiefe einer Liste:
(defun tiefe (liste)
  (cond ((null liste) 1)
        ((atom liste) 0)
        (t (+ (apply #'max[1](mapcar #'tiefe liste)) 1))))
```

MAPCAR und MAPCAN können zur Formulierung von Filter-Funktionen verwendet werden. Filter-Funktionen sind Funktionen, die aus einer Liste alle Elemente entfernen, die einer best. Bedingung genügen/nicht genügen. In den meisten Fällen ist MAPCAN MAPCAR vorzuziehen, da MAPCAR für jedes Objekt, auf das das Filterprädikat nicht zutrifft, in der Resultatliste ein NIL zurückläßt.

[1]Die Funktion MAX akzeptiert eine Liste von Zahlen als Argument und evaluiert zu der größten in dieser Liste enthaltenen Zahl.

Beispiel (7-4)

```
> (defun our-remove-if-not1 (test liste)
    (mapcar #'(lambda (x) (when (funcall test x) x)) liste))
  OUR-REMOVE-IF-NOT1
> (defun our-remove-if-not2 (test liste)
    (mapcan #'(lambda (x) (when (funcall test x) (list x))) liste))
  OUR-REMOVE-IF-NOT2
> (our-remove-if-not1 #'numperp '(a 6 (2 3) 99))
  (NIL 6 NIL 99)
> (our-remove-if-not2 #'numperp '(a 6 (2 3) 99))
  (6 99)
```

7.2 Mapping auf (Rest)Listen

Die beiden Funktionen MAPLIST und MAPCON wenden nicht eine Funktion nacheinander auf alle Elemente der Argumentliste (bzw. auf die i-ten Elemente der n Argumentlisten) an, sondern zunächst auf die ganze Argumentliste(n), dann auf die Restliste(n) der Argumentliste(n), auf die Restliste(n) dieser Liste(n), ... etc bis das Ende der (kürzesten) Liste erreicht ist:

(funktion $liste_1$ ... $liste_n$*)*
(funktion (rest $liste_1$*)* ... *(rest* $liste_n$*))*
(funktion (rest (rest $liste_1$*))* ... *(rest (rest* $liste_n$*)))*
⋮ ⋮ ⋮

MAPLIST	*Funktion* *Liste**	[Funktion]
Die Funktion MAPLIST wendet *Funktion* zunächst auf die Argumentlisten und dann nacheinander auf den REST dieser Listen an und liefert als Wert die Liste aller Werte. Wenn die Listen nicht dieselbe Länge besitzen, terminiert die Iteration, nachdem die kürzeste Liste abgearbeitet ist.		

MAPCON	*Funktion* *Liste**	[Funktion]
Die Funktion MAPCON unterscheidet sich von MAPLIST nur dadurch, daß die Ergebnisse der Funktion mit NCONC verknüpft werden.		

Die Beziehung zwischen MAPLIST und MAPCON entspricht der zwischen MAPCAR und MAPCAN; d.h. MAPCON läßt sich definieren als:

$$<\text{mapcon } f\ x_1 \ldots x_n> \equiv <\text{apply \#'nconc } <\text{maplist } f\ x_1 \ldots x_n>>$$

Die Funktion MAPL ist das *listenorientierte* Gegenstück zu MAPC:

MAPL	*Funktion Liste**	[Funktion]
Die Funktion MAPL verhält sich zu MAPLIST wie MAPC zu MAPCAR: Sie arbeitet wie MAPLIST, akkumuliert aber nicht die aus den Funktionsanwendungen resultierenden Werte.		

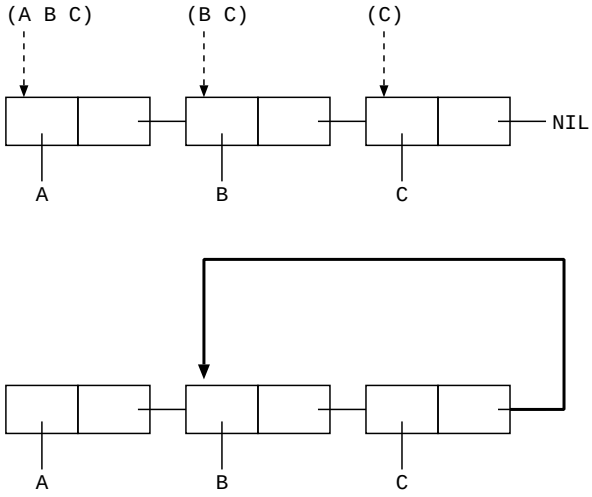
Beispiel (7-5)

```
;;; Die Verwendung der Identitätsfunktion läßt die Arbeitsweise
;;; von MAPLIST besonders deutlich werden:
;;; X wird nacheinander an (A B C D), (B C D), (C D) und (D)
;;; gebunden.
> (maplist #'(lambda (x) x) '(a b c d))
  ((A B C D) (B C D) (C D) (D))
> (maplist #'first '(a b c d))
  (a b c d)
> (maplist #'cons '(a b c) '(1 2 3))
  (((A B C) 1 2 3) ((B C) 2 3) ((C) 3))
> (maplist #'(lambda (x) (apply #'* x)) '(1 2 3 4 5))
  (120 120 60 20 5)
> (mapcon #'(lambda (x) (list x)) '(a b c d))
  ((A B C D) (B C D) (C D) (D))
> (mapcon #'reverse '(a (b c) (d)))
  ((D) (B C) A (D) (B C) (D))
> (mapl #'(lambda (x) (apply #'* x)) '(1 2 3 4 5))
  (1 2 3 4 5)

> (mapl #'(lambda (x) (print (apply #'* x))) '(1 2 3 4 5))
  120      Seiteneffekt
  120      Seiteneffekt
  60       Seiteneffekt
  20       Seiteneffekt
```

```
   5          Seiteneffekt
   (1 2 3 4 5)      Wert
;;;  Die Verwendung von destruktiven Funktionen erfordert besondere
;;;  Sorgfalt, wie das folgende Beispiel verdeutlicht:
> (mapcon #'(lambda (x) (print x)) '(a b c))
   (A B C)                                          Seiteneffekt
   (B C)                                            Seiteneffekt
   (C)                                              Seiteneffekt
   (A B C B C B C B C B C B C B C B C B C . . . )    zirkuläre Liste
```

Da NCONC anderes als APPEND seine Argumente nicht kopiert, wird bei der Verknüpfung einer Folge von Listen $L_1 \ldots L_n$ der auf NIL weisende End-Pointer von L_i so 'verbogen', daß er auf das erste Element von L_{i+1} weist. Der Versuch, die Liste (A B C) mit (REST (A B C)) und (REST (REST (A B C))) zu verknüpfen, führt dazu, daß der End-Pointer nicht mehr auf NIL, sondern auf B, das erste Element der zweiten Liste, die eine Teilliste der ersten Liste ist, weist. Der Versuch, den End-Pointer der zweiten Liste umzusetzen, scheitert, da die Liste mittlerweile zirkulär ist.



Kapitel 8

Property- und Assoziationslisten

Es gibt verschiedene Möglichkeiten, innerhalb eines Programms Objekte, die durch bestimmte Eigenschaften charakterisiert sind, zu repräsentieren. Da Listen die zentrale Datenstruktur von LISP bilden, ist es naheliegend, zu diesem Zweck Listen zu verwenden. Zwei spezielle Listentypen eignen sich hierzu besonders: Property- und Assoziationslisten. Ein entscheidender Unterschied zwischen beiden Typen von Listen besteht darin, daß Property-Listen in einem gleich zu spezifizierenden Sinn an Symbole gebunden sind, während Assoziationslisten beliebig generiert und manipuliert werden können.

8.1 Propertylisten

LISP-Symbole sind komplexe Objekte, die aus *print-name*, *value-binding function-binding*, *package-cell* und *property-list* bestehen (vgl. Kapitel 1). Die Property-Liste (kurz: P-Liste) eines Symbols besteht aus einer (möglicherweise leeren) Folge von Einträgen, wobei ein Eintrag aus einem Schlüssel/Indikator und einem diesem Indikator zugeordneten Wert besteht. Die P-Liste neu erzeugter Symbole ist zunächst leer. Da die Generierung einer P-Liste Teil der Generierung eines Symbols ist, gibt es keinen speziellen Generator für P-Listen; d.h. wir können uns auf die Beschreibung von Selektoren und Modifikatoren für P-Listen beschränken.

8.1.1 Selektoren

Für jeden Teil eines Symbols gibt es einen Selektor: SYMBOL–FUNKTION, SYMBOL–VALUE, SYMBOL–NAME, SYMBOL–PACKAGE und SYMBOL–PLIST:

SYMBOL-PLIST	*Symbol*	[Funktion]
Die Funktion SYMBOL-PLIST nimmt ein Symbol *Symbol* als Argument und liefert als Wert die Property-Liste von *Symbol*.		

Um nicht die komplette P-Liste, sondern den Wert für ein Attribut der P-Liste zu erhalten, verwendet man die Funktion GET:

GET *Symbol Schlüssel* [Funktion]
GET sucht nach einem Indikator in der Property-Liste von *Symbol*, der mit *Schlüssel* identisch ist (EQ). Gibt es einen derartigen Schlüssel, wird der ihm zugeordnete Wert zurückgegeben; sonst NIL.

8.1.2 Modifikatoren

Es gibt zwei Modifikatoren für P-Listen: PUTPROP[1] generiert einen neuen bzw. aktualisiert einen bestehenden Eintrag der P-Liste eines Symbols, und REMPROP entfernt einen Eintrag (Attribut & Wert) aus der P-Liste eines Symbols:

PUTPROP *Symbol Wert Schlüssel* [Funktion]
Durch PUTPROP wird die Property-Liste von *Symbol* durch das Paar *Schlüssel Wert* ergänzt; wenn *Schlüssel* in der Property-Liste bereits enthalten ist, wird der alte Wert durch *Wert* ersetzt. Die PUTPROP-Form evaluiert zu *Wert*.

REMPROP *Symbol Schlüssel* [Funktion]
REMPROP sucht nach einem Indikator in der Property-Liste von *Symbol*, der mit *Schlüssel* identisch ist (EQ). Gibt es einen derartigen Schlüssel, dann werden er und der ihm zugeordnete Wert aus der Property-Liste von *Symbol* entfernt. REMPROP evaluiert zu zu NIL.

Beispiel (8-1)

```
> (symbol-plist 'hans)
  NIL
> (setf (symbol-plist 'hans) '(arbeit selten interessen keine
   schulden hoch))
  (ARBEIT SELTEN INTERESSEN KEINE SCHULDEN
```

[1] PUTPROP ist nicht in allen Common LISP Implementierungen verfügbar. In diesem Fall ist eine SETF-GET Kombination zu verwenden, um einen neuen Eintrag zu generieren; bzw. PUTPROP zu definieren: (defun putprop (symbol wert schluessel) (setf (get symbol schluessel) wert)).

```
    HOCH)
> (get 'hans 'interessen)
  KEINE
> (get 'hans 'freundin)
  NIL
> (putprop 'hans 'anna 'freundin)
  ANNA
> (putprop 'hans '(gustav elfriede) 'eltern)
  (GUSTAV ELFRIEDE)
> (get 'hans 'eltern)
  (GUSTAV ELFRIEDE)
> (remprop 'hans 'interessen)
  NIL
> (get 'hans 'interessen)
  NIL
> (symbol-plist 'hans)
  (ELTERN (GUSTAV ELFRIEDE) FREUNDIN ANNA
   ARBEIT SELTEN SCHULDEN HOCH)
```

8.2 Assoziationslisten

Eine Assoziationsliste (kurz: *A-Liste*) ist eine Liste, deren Elemente Paarlisten sind, die wir im folgenden als *Einträge* bezeichnen. Das erste Element jedes Eintrags repräsentiert den Indikator oder Schlüssel eines *Attribut–Wert* Paares, der Rest der Liste den diesem Schlüssel zugeordneten (*assoziierten*) Wert[2]: ((*Schlüssel*$_1$ *Wert*$_1$) (*Schlüssel*$_2$ *Wert*$_2$) ... (*Schlüssel*$_n$ *Wert*$_n$)).

Für Assoziationslisten gibt es spezielle Selektorfunktionen, die es erlauben, auf Einträge zuzugreifen, die einen spezifischen Schlüssel oder Wert besitzen. A-Listen unterscheiden sich in zwei Punkten wesentlich von P-Listen:

1. Anders als bei den P-Listen kann in einer Assoziationsliste ein Schlüssel mehrfach vorkommen.

2. Während die Modifikation von P-Listen *destruktiv* ist, sind diese Operationen auf A-Listen in der Regel *nicht-destruktiv* ist, da beim Einfügen neuer Einträge bzw. Entfernen von Einträgen nicht die alte A-Liste verändern, sondern eine neue A-Liste erzeugt wird.

[2]Diese Darstellung ist strenggenommen nur dann korrekt, wenn es sich bei den Paarlisten um *dotted pairs* handelt. Verwendet man statt Paarlisten (im Sinne von *dotted pairs*) Listen mit zwei Elementen, enthält der Rest der Liste den zugeordneten Wert.

ASSOC	*Schlüssel* *A-Liste*	[Funktion]

Die Funktion ASSOC liefert als Wert den ersten Eintrag aus *A-Liste*, dessen CAR mit *Schlüssel* übereinstimmt; sonst NIL. Der Vergleich wird mit EQL durchgeführt.

RASSOC	*Schlüssel* *A-Liste*	[Funktion]

Die Funktion RASSOC liefert als Wert den ersten Eintrag von *A-Liste*, dessen CDR mit *Schlüssel* übereinstimmt; sonst NIL. Der Vergleich wird mit EQL durchgeführt.

Bei beiden Funktionen ist es durch Verwendung des TEST-*keyword* möglich, andere Identitätsfunktionen zu verwenden. Neben ASSOC und RASSOC gibt es noch die Funktionen ASSOC-IF und ASSOC-IF-NOT bzw. RASSOC-IF und RASSOC-IF-NOT, die es ermöglichen, ein Prädikat anzugeben, durch das festgelegt werden kann, unter welcher Bedingung ein Eintrag auszuwählen ist. Auerdem ist eine ASSOC/RASSOC-Form ein zulässiges Argument für SETF: *(setf (assoc ...) ...)*

Beispiel (8-2)

```
> (setq *personen* '((anna schmidt) (kurt masters) (judith klein)
                     (kurt krenz)))
  ((ANNA SCHMIDT) (KURT MASTERS) (JUDITH KLEIN)
   (KURT KRENZ))
> (assoc 'kurt *personen*)          Zugriff über den Vornamen.
  (KURT MASTERS)
 > (rassoc 'masters *personen*)     Zugriff über den Nachnamen.
  NIL                               Aber: Es gibt keine Paarliste
                                    α, mit REST(α) = MASTERS.
 > (rassoc '(masters) *personen*)   Ein besserer Versuch. Doch
  NIL                               RASSOC/ASSOC verwenden EQL!
> (rassoc '(masters) *personen* :test #'equal)
  (KURT MASTERS)
> (setf (second (assoc 'anna *personen*)) 'masters)
  MASTERS
> *personen*
  ((ANNA MASTERS) (KURT MASTERS) (JUDITH KLEIN)
   (KURT KRENZ))
> (rassoc '(masters) *personen* :test #'equal)
```

(ANNA MASTERS)

Um statt des kompletten Eintrags nur den Wert zu erhalten, mu nur das zweite Element des Eintrags, den ASSOC liefert, mit SECOND ausgewhlt werden. Definiert man zu diesem Zweck eine eigene Funktion, so sollte sie nicht zu NIL evaluieren, wenn es fr den als Parameter bergebenen Schlssel keinen Eintrag in der Liste gibt. Anderenfalls ist NIL als Funktionswert mehrdeutig. Es kann bedeuten: (a) NIL ist in der Liste als Wert des angegebenen Schlssels spezifiziert oder (b) es gibt in der Liste keinen Eintrag fr ihn.

;;;
;;; Die Funktion WERT liefert den einem Schlssel in einer Assoziationsliste zugeordneten Wert.

```
(defun wert (symbol a-liste)
  (let ((eintrag (assoc symbol a-liste)))
    (if eintrag
        (second eintrag)
        '?)))
```

```
> (wert 'kurt *personen*)
  masters
> (wert 'mira *personen*)
  ?
```

In vielen Fllen ist es ntzlich, alle Schlssel bzw. alle Werte aus einer Assoziationsliste extrahieren zu knnen. Die uniforme Struktur dieser Listen legt es nahe, zu diesem Zweck eine MAPxxx-Funktion zu verwenden:

;;;
;;; Die Funktion ALLE-SCHLUESSEL liefert die Liste aller Schlüssel einer verwendeten Schlüssel.

```
(defun alle-schluessel (a-liste)
  (mapcar #'first a-liste))
```

;;;
;;; Die Funktion ALLE-WERTE liefert die Liste aller Schlüssel einer Assoziationsliste verwendeten W

```
(defun alle-werte (a-liste)
   (mapcar #'second a-liste))
```

Aus unserer Namensliste selektiert also ALLE-SCHLUSSEL die Vor- und ALLE-WERTE die Nachnamen:

```
> (alle-schluessel *personen*)
  (anna kurt judith kurt)
```

```
> (alle-werte *personen*)
  (MASTERS MASTERS KLEIN KRENZ)
```

Kapitel 9

Ein- und Ausgabe

In diesem Kapitel erläutern wir zunächst kurz das in den meisten modernen Programmiersprachen zur Realisierung der Ein- und Ausgabeoperationen verwendete *Stream*-Konzept und beschreiben dann die Funktionen, die Common LISP für diese Zwecke zur Verfügung stellt. Dazu gehören Funktionen, die es erlauben, einzelne Objekte einzulesen bzw. auszugeben, eine Funktion zur Erzeugung beliebig formatierter Ausgaben und Funktionen, die den Zugriff (Schreiben/Lesen) auf Textdateien ermöglichen.

9.1 Streams

In Common-LISP arbeiten alle Ein-/Ausgabeoperationen auf *Streams*. Ein Stream ist ein LISP-Objekt, das als Produzent/Konsument von Daten dient. Seine primäre Funktion liegt darin, eine Verbindung zwischen den Ein-/Ausgabeoperationen innerhalb des LISP-Systems und den externen Ein-/Ausgabegeräten (Terminal, Drucker, ...) und Dateien herzustellen.

Abbildung (9-1)

Alle Ein-/Ausgabefunktionen erlauben als optionalen Parameter die Angabe eines Stream-Namens, der den Stream bestimmt, in den die Daten geschrieben bzw. aus dem sie gelesen werden sollen. Fehlt diese Angabe, wird der Stream verwendet, der der Variable *standard-input* bzw. *standard-output* zugeordnet ist. Standardwert ist in beiden Fällen der Stream, der dem Terminal zugeordnet ist; d.h. Eingaben werden von der Tastatur erwartet und Ausgaben auf den Bildschirm gelenkt. Abhängig von den auf ihnen ausführbaren Operationen werden Streams als *Eingabe-*, *Ausgabe-* und *bidirektionale* Streams bezeichnet.

9.2 Einfache Ein- und Ausgabeoperationen

Alle der folgenden Funktionen können wegen des Streamkonzepts sowohl auf dem Terminal wie auch auf beliebigen Dateien operieren.

9.2.1 Ausgabefunktionen

PRINT	*Objekt* [*Ausgabe-Stream*]	[Funktion]
PRINT schreibt eine graphische Repräsentation von *Objekt* in den aktuellen bzw. den spezifizierten *Ausgabe-Stream* (Seiteneffekt). Vor der Ausgabe der Repräsentation des Objekts erfolgt ein Zeilenvorschub; anschließend wird ein Leerzeichen ausgegeben. PRINT evaluiert zu *Objekt*.		

PRINC	*Objekt* [*Ausgabe-Stream*]	[Funktion]
PRINC schreibt eine graphische Repräsentation von *Objekt* in den aktuellen bzw. den spezifizierten *Ausgabe-Stream* (Seiteneffekt). Es werden keine Escape-Zeichen ausgegeben. Symbole werden durch ihren Print-Namen repräsentiert; Strings ohne begrenzende Anführungszeichen ausgegeben. PRINC evaluiert zu *Objekt*.		

TERPRI	[*Ausgabe-Stream*]	[Funktion]
Diese Funktion bewirkt einen Zeilenvorschub (Seiteneffekt) und evaluiert zu NIL.		

Da der *Reader* und der *Printer* - das sind die Teile des LISP-Interpreters, die für das Einlesen bwz. die Ausgabe von Objekten zuständig sind - nicht vollkommen standardisiert sind, können die durch Ein-/Ausgabeoperationen erzielten Resultate leicht differieren:

Beispiel (9-1)

Allegro Common LISP
MS-DOS

```
> (print 88.7)
88.7
88.7
> (princ 88.7)88.7
88.7
```

VAX-LISP
VMS

```
> (print 88.7)
88.7
88.7
> (princ 88.7)
88.7
88.7
```

```
> (print 'abc)
  ABC
  ABC
> (princ 'abc)ABC
  ABC

> (print "Hallo!")
  "Hallo!"
  "Hallo!"
> (princ "Hallo!")Hallo!
  "Hallo!"

> (print 'R\(1\))
  |R(1)|
  |R(1)|
> (princ 'R\(1\))R(1)
  |R(1)|

> (terpri)

  NIL
> (progn
      (print 4)
      (print 5))
  4
  5
  5
> (progn
      (princ 4)
      (princ 5))45
  5

> (progn
      (princ 4)
      (print 5))4
  5
  5

> (progn
```

```
> (print 'abc)
  ABC
  ABC
> (princ 'abc)
  ABC
  ABC
> (print "Hallo!")
  "Hallo!"
  "Hallo!"
> (princ "Hallo!")
  Hallo!
  Hallo!
> (print 'R\(1\))
  |R(1)|
  |R(1)|
> (princ 'R\(1\))
  R(1)
  R(1)
> (terpri)
  NIL

> (progn
      (print 4)
      (print 5))
  4
  5
  5
> (progn
      (princ 4)
      (princ 5))
  45
  5
> (progn
      (princ 4)
      (print 5))
  4
  5
  5
> (progn
```

```
    (print 4)
    (princ 5))
4 5
5
```

```
    (print 4)
    (princ 5))
4 5
5
```

```
;;;
;;;
;;; Die Funktion PRINT-STRUKTUR nimmt ein beliebiges LISP-Objekt als Argument und gibt
;;; Die Struktur von Listen wird durch Einrücken der Elemente relativ zu ihrer Verschachtelungs


;;; Hauptfunktion
(defun print-struktur (ausdruck)
   (print-at ausdruck 0))
```

```
;;;
;;; Die Funktion PRINT-AT nimmt ein LISP-Objekt als Argument und gibt es auf dem Bildschirm in S

(defun print-at (ausdruck spalte)
   (cond ((atom ausdruck)                ; Atomares Argument:
          (tab spalte)                   ; Bewege den Cursor in die richtige Spalte.
          (princ ausdruck)               ; Gebe AUSDRUCK aus.
          (terpri))                      ; Beginne eine neue Zeile.
         (t (print-at "(" spalte)        ; Liste: Schreibe öffnende Klammer
            (dolist (x ausdruck)         ; Gebe Listenelement aus.
              (print-at x (+ spalte 2)))
              (print-at ")" spalte))))   ; Schreibe schließende Klammer.

;;; TAB gibt n Leerzeichen aus.
(defun tab (n)
   (dotimes (x n)
     (princ " ")))


> (print-struktur '(a () (b (c d))))
  (
    A
    NIL
    (
       B
       (
          C
          D
       )
    )
  )
```

Neben PRINT und PRINC, die beliebige LISP-Objekte als Argumente akzeptieren, gibt es noch eine Reihe von Funktionen für spezielle Typen von Objekten:

WRITE-CHAR *Zeichen* [*output-stream*] [Funktion]
WRITE-CHAR schreibt das *Zeichen* korrespondierende Zeichen in den aktuellen Ausgabestream bzw. *output-stream* und evaluiert zu *Zeichen*.

WRITE-STRING *String* [*output-stream*] [Funktion]
WRITE-STRING schreibt die Folge von Zeichen, aus denen *String* sich zusammensetzt in den aktuellen Ausgabestream bzw. *output-stream* und evaluiert zu *String*.

WRITE-LINE *String* [*output-stream*] [Funktion]
WRITE-LINE verhält sich wie WRITE-STRING mit der Ausnahme, daß anschließend ein Zeilenvorschub erfolgt.

Beispiel (9-2)

```
> (write-char #\a)
  a
  #\a
> (progn (write-char #\a) (write-char #\space) (write-char #\z))
  a z
  #\z
> (write-string "a")
  a
  "a"
> (write-string "Test(1)")
  Test(1)
  "Test(1)"
> (progn (write-char #\a) (write-string " ...") (write-char #\z))
  a ... z
  #\z
> (progn (write-char #\a) (write-line " ...") (write-char #\z))
  a ...
  z
  #\z
```

9.2.2 Eingabefunktionen

Die READ-Funktion akzeptiert beliebige LISP-Objekte; READ-CHAR liest ein Zeichen und READ-LINE eine Folge von Zeichen ein, aus denen dann ein String gebildet wird.

READ [*Eingabe-Stream* [*EOF-Fehler-P* [*EOF-Wert*]]] [Funktion]
Die Funktion READ liest die geschriebene Repräsentation eines Lisp-Objekts aus dem Standardeingabestream bzw. *Eingabe-Stream* ein, generiert ein korrespondierendes Objekt und liefert dieses Objekt als Wert.

READ-CHAR [*Eingabe-Stream* [*EOF-Fehler-P* [*EOF-Wert*]]] [Funktion]
READ-CHAR liest ein Zeichen aus dem Standardeingabestream bzw. *Eingabe-Stream* ein und liefert als Wert das diesem Zeichen entsprechende Zeichenobjekt.

READ-LINE [*Eingabe-Stream* [*EOF-Fehler-P* [*EOF-Wert*]]] [Funktion]
READ-LINE liest eine Zeile aus dem Standardeingabestream bzw. *Eingabe-Stream* ein. Das Zeilenende muß durch ein *Newline*-Zeichen markiert sein. Als erster Wert wird die Zeile als String zurückgegeben; als zweiter Wert NIL gdw. die Eingabe normal beendet wurde bzw. T wenn sie durch Erreichen der EOF-Marke terminiert wurde.

Beispiel (9-3)

```
> (read)77   <RETURN>          Ein Objekt kann direkt eingeben werden
  77
> (read)     <RETURN>          oder nach einem RETURN
  77         <RETURN>
  77
> (read)     <RETURN>          oder nach mehreren RETURN
             <RETURN>          READ terminiert erst, nachdem ein
             <RETURN>          vollständiges LISP-Objekt eigelesen wurde.
  77         <RETURN>
  77
> (read)kurt
  KURT
> (read)"Ein alter Hase"
  "Ein alter Hase"
```

```
> (read)(a b () c)
  (A B NIL C)
> (eq '(a b c) (read))(a b c)
  NIL
> (equal '(a b c) (read))(a b c)
  T
> (read-char)U
  #\U
> (read-char)4
  #\4
> (read-char)<RETURN>
  #\newline
> (read-char) <RETURN>
  #\space
> (read-char)KURT
  #\K
> error: unbound variable - URT
  if continued: try evaluating symbol again
  1>
   [ back to top level ]
> (READ-CHAR)"A"
  #\"
> error: unbound variable - A
  if continued: try evaluating symbol again
  1>
   [ back to previous break level ]
> (read-line)Ein alter Mann
  "Ein alter Mann"
  NIL
> (read-line)"Ein alter Depp"
  ""Ein alter Depp""
  NIL
> (read-line) <RETURN>
  ""
  NIL
> (read-line)HALLO
  "HALLO"
  NIL
```

> (read-line) 77 Osterhasen
" 77 Osterhasen"
NIL
> (read-line)(a b c)
"(a b c)"
NIL

9.3 Formatierte Ausgabe

Mit den bislang eingeführten Ausgabefunktionen ist es relativ umständlich, Informationen in einem ansprechenden Format auszugeben. Für diese Zwecke gibt es die sehr mächtige FORMAT-Funktion:

FORMAT *Ziel Kontroll–String Argument** [Funktion]
FORMAT erzeugt eine formatierte Ausgabe des *Kontroll-Strings*. Alle Zeichen des Strings, mit Ausnahme der in ihm enthaltenen Formatanweisungen, werden (ohne Anführungszeichen) an der durch *Ziel* spezifizierten Stelle ausgegeben. Wenn *Ziel* NIL ist, liefert FORMAT als Wert den formatierten String; in allen anderen Fällen erfolgt seine Ausgabe als Seiteneffekt in den durch *Ziel* spezifizierten Stream und FORMAT evaluiert zu NIL. Wenn *Ziel* T ist, wird der String in den Standard-Ausgabestream geschrieben.

Zu den wichtigsten FORMAT-Anweisungen gehören die folgenden:

~A	:	Ausgabe des nächsten Arguments unter Verwendung von PRINC;
~S	:	Ausgabe des nächsten Arguments unter Verwendung von PRINT;
~%	:	Zeilenumbruch;
~*spalte*T	:	Ausgabe der Zahl von Leerzeichen, die erforderlich ist, um den Cursor auf der Spalte *spalte* zu positionieren. Befindet er sich bereits an bzw. jenseits dieser Position, ist diese Anweisung wirkungslos.

Beispiel (9-4)

> (format t "Nur keine Panik!")
Nur keine Panik!
NIL
> (format nil "Nur keine Panik!")

```
  "Nur keine Panik!"
> (format t "~%Nur keine Panik!")

  Nur keine Panik!
  NIL
> (format t "~%Nur~%keine~%Panik!")

  Nur
  keine
  Panik!
  NIL
> (format t "Der Wert von~S ist:~S." '(+ 3 4) (+ 3 4))
  Der Wert von (+ 3 4) ist: 7.
  NIL
> (progn (format t "Der Wert von~S ist:~40T~S.~%" '(+ 3 4) (+ 3 4))
         (format t "Der Wert von~S ist:~40T~S." 'pi pi))
  Der Wert von (+ 3 4) ist:       7.
  Der Wert von PI ist:            3.141592.
  NIL
```

9.4 Ein- und Ausgabeoperationen auf Dateien

Um Informationen aus einer Datei einzulesen bzw. in eine Datei zu schreiben, muß diese Datei zunächst geöffnet werden:

OPEN	*Dateiname* *&key :direction*	[Funktion]
OPEN liefert als Wert einen Stream, der mit der durch *Dateiname* bezeichneten Datei verbunden ist. Bei *Dateiname* kann es sich um ein Symbol, einen Pfadnamen, einen String oder einen Streamnamen handeln. Das *keyword*-Argument spezifiziert, welcher Typ von Stream erzeugt wird.		

CLOSE	*Streamname*	[Funktion]
Der mit *Streamname* bezeichnete Stream wird geschlossen. Es können anschließend keine weiteren Ein- oder Ausgabeoperationen auf ihm ausgeführt werden.		

Wenn eine Datei geöffnet wird [OPEN], wird ein Stream erzeugt, der das LISP-System und das Dateisystem miteinander verknüpft: Den Operationen auf dem Stream (*Lesen*, *Schreiben*) korrespondieren Operationen auf der Datei. Wenn der Stream geschlossen wird [CLOSE], wird damit auch die assoziierte Datei geschlossen und die Verbindung von LISP-System und Dateisystem wieder aufgehoben. Es sollte unbedingt darauf geachtet werden, alle geöffneten Dateien vor Verlassen des LISP–Systems wieder zu schließen. Enthält eine Datei ausschließlich LISP-Formen (Programmdatei), kann sie direkt eingelesen werden:

LOAD	*Dateiname*	[Funktion]
Durch diese Funktion wird die mit *Dateiname* (String) bezeichnete Datei geladen, und die in ihr enthaltenen Formen werden evaluiert.		

Beim Einlesen werden die in der Datei enthaltenen LISP-Formen evaluiert; d.h. die in der Datei definierten Funktionen können sofort verwendet werden.

Beispiel (9-5)

```
> (setf my-stream (open "schrott.lsp" :direction :output))
  #<File-Stream: 4c4d3a4c>
> (print '(+ 88 7) my-stream)
  (+ 88 7)
> (print '(print (+ 88 7)) my-stream)
  (print (+ 88 7))
> (print "Ende der Datei." my-stream)
  "Ende der Datei."
> (close my-stream)
NIL
> (print 'geht_es_noch? my-stream)
  error: file not open
  1> [return to top level]
> (load "my-stream") ; beim Laden einer Datei werden alle in ihr
  95 ; enthaltenen LISP-Formen evaluiert
  T
> (setf my-stream (open "schrott.lsp" :direction :input)
  #<File-Stream: 4c4d384e>
> (read my-stream)
  (+ 88 7)
> (read my-stream)
  (print (+ 88 7))
> (read my-stream)
```

```
  "Ende der Datei."
> (read my-stream)
  EOF on stream: #<File-Stream: 4c4d384e>
  1>
```

Wie das vorangegangene Beispiel zeigt, führt nach dem Erreichen des Dateiendes ein weiterer Lesezugriff auf diese Datei zu einem Fehler. Die optionale Spezifikation eines *EOF-Wertes* in dem Fall, daß der Lesezugriff auf eine Datei erfolgt, erlaubt es, diese Fehlermeldung zu unterdrücken: Beim Erreichen des Dateiendes wird nun der *EOF-Wert* zurückgegeben. Es ist sinnvoll, als EOF-Wert nicht NIL, sondern ein Symbol zu verwenden, von dem man weiß, daß es nicht in der Datei vorkommt (z.B. EOF).

Das explizite Öffnen und Schließen einer Datei mit Hilfe von OPEN und CLOSE kann man vermeiden, wenn man alle Schreib- und Leseoperationen innerhalb einer WITH-OPEN-FILE-Form realisiert:

WITH-OPEN-FILE (*Str Dateiname &key :dir*) {*Form*}* [Makro]
Eine WITH-OPEN-FILE-Form erzeugt den Stream *Str*, der der mit *Dateiname* bezeichneten Datei zugeordnet wird. Anschließend werden alle Formen des Anweisungsbocks nacheinander evaluiert (implizites PROGN). Das Verlassen des Anweisungsblocks führt in jedem Fall dazu, daß die zuvor geöffnete Datei geschlossen wird.

Eine gleichzeitige Bearbeitung mehrerer Dateien, um z.B. Daten aus einer Datei zu lesen und in einer zweiten zu speichern, läßt sich realisieren, indem eine WITH-OPEN-FILE-Form in eine andere eingebettet wird:

```
... (with-open-file (stream1 datei1 :direction :input)
      (with-open-file (stream2 datei2 :direction :output)
        [Lese Wert aus datei1]
        [Verarbeite Wert]
        [Schreibe Resultat in datei2] )) ...
```

Ein weiterer großer Vorzug von WITH-OPEN-FILE liegt darin, daß das System auch dann versucht, geöffnete Dateien zu schließen, wenn die Evaluierung einer der eingebetteten Formen einen Fehler generiert.

Beispiel (9-6)

```
;;;; (A) Generieren einer Genusliste aus einer Datei
;;; READ-NOMEN
;;; Argument : Ein Dateiname datei.
;;; Die Funktion nimmt als Argument eine Datei, die neben Kommentaren ausschließlich
;;; Sätze des folgenden Formats enthält: 'Das Genus von xxx ist yyy'.
;;; Die Funktion liest alle Angaben der Datei und bildet eine Liste der Form:
;;; ((Nomen_1 Genus_1) ... (Nomen_n Genus_n)), mit Genus ∈ {m, f, n, ?}.
```

```
(defun read-nomen (datei)
  (with-open-file (my-stream datei :direction :input)
    (do ((nomen () (append nomen (list (fourth satz) (auswerten (sixth satz)))))
         (satz (read-satz my-stream 6) (read-satz my-stream 6)))
        ((eq satz 'eof) nomen))))
```

```
;;; READ-SATZ
;;; Argumente : Ein Stream stream und eine Zahl satzlaenge.
;;; Die Funktion liest eine durch satzlaenge bestimmte Anzahl von Ausdrücken aus stream
;;; und liefert als Wert eine Liste, die die Ausdrücke enthält.
(defun read-satz (stream satzlaenge)
  (let ((wort ()) (satz ()))
    (dotimes (x satzlaenge satz)
      (setq wort (list (read stream nil 'eof)))
      (if (eq wort 'eof)
          (return 'eof)
          (setf satz (append satz (list (read stream nil 'eof))))))))

;;; AUSWERTEN
;;; Argument : Ein Symbol wort.
;;; Die Funktion 'übersetzt' einen Genusbezeichner wie Maskulinum, Femininum, ..., etc.
;;; in eine gewünschte Abkürzung.
(defun auswerten (wort)
  (case wort
    (femininum 'f)
    (maskulinum 'm)
    (neutrum 'n)
    (t '?)))
```

```
;;;; (B) Suche nach einem Begriff in einer Datei

;;; SUCHE-ZEILE
;;; Argumente : Ein String teilstring und ein Dateinamen datei.
;;; Die Funktion sucht in datei nach einer Zeile, die teilstring enthält und gibt diese Zeile
;;; auf dem Bildschirm aus.
(defun suche-zeile (teilstring datei)
  (with-open-file (my-stream datei :direction :input)
    (do ((zeile (read-line my-stream nil 'eof)
```

```
        (read-line my-stream nil 'eof)))
((eq zeile EOF)
 (format t "Keine Zeile gefunden, die ~A enthaelt!" teilstring))
(when (search teilstring zeile)
   (format t " ~% ~A" zeile)
   (return t)))))
```

```
;;; SEARCH
;;; Argumente : Zwei Strings str1 und str2.
;;; Die Funktion überprüft, ob der String str1 Teilstring von str2 ist. Als Wert wird
;;; der Index des linken Zeichens des ersten Teilstrings von str2 zurückgegeben, der mit
;;; str1 übereinstimmt; sonst NIL. Unterschiede bzgl. Groß- und Kleinschreibung werden
;;; vernachlässigt.
(defun search (str1 str2)
  (dotimes (x (- (length str2) (- (length str1) 1)))
    (when (string-equal[1]str1 (subseq[2]str2 x (+ x (length str1))))
      (return x))))
```

[1]Die Funktion STRING-EQUAL evaluiert zu T gdw. sich die beiden miteinander zu vergleichenden Strings höchstens bzgl. Groß- und Kleinschreibung einzelner Buchstaben unterscheiden; sonst NIL.

[2]Die Funktion SUBSEQ *String Anfang Ende* liefert den durch *Anfang* und *Ende* markierten Teilstring von *String* als Wert.

Kapitel 10

Endliche Automaten

Endliche Automaten[1] bilden den einfachsten Typ abstrakter Maschinen, die zur Bearbeitung computerlinguistischer Aufgabenstellungen verwendet werden. Sie sind einfach zu implementieren und erlauben eine effiziente Lösung für eine vielzahl von linguistischer Fragestellungen. Allerdings ist ihre Leistungsfähigkeit begrenzt: Aus diesem Grund werden endliche Automaten vor allem im Bereich der Phonologie und Morphologie (vereinzelt auch in Syntax und Semantik) verwendet.

10.1 Grundlegende Konzepte

Endliche Automaten sind abstrakte Maschinen, die Zeichen- bzw. Symbolfolgen verarbeiten und diese akzeptieren oder zurückweisen. Es gibt zwei Typen von endlichen Automaten: nicht-deterministische Automaten (NEA) und deterministische Automaten (DEA), die eine spezielle Teilklasse der NEAs bilden.

Definition (10-1) *Nicht-deterministischer endlicher Automat*

Ein *nicht-deterministischer endlicher Automat* $M = <Q, \Sigma, \sigma, q_0, F>$ besteht aus:

1. einer endlichen Menge von Zuständen $Q = \{q_0, q_1, \ldots, q_n\}$;
2. einer endlichen Menge $\Sigma = \{w_1, \ldots, w_m\}$ von Eingabesymbolen;
3. der Übergangsfunktion σ:$Q \times (\Sigma \cup \{e\}) \rightarrow Pot(Q)$[2];
4. dem Anfangszustand $q_0 \in Q$ und
5. einer Menge F (F ist eine Teilmenge von Q) von Endzuständen.

[1] Endliche Automaten werden in der englischsprachigen Literatur als *finite state automata* (FSA) bezeichnet.

[2] Die Übergangsfunktion σ ist eine *partielle* Funktion; d.h. sie muß nicht für alle $<q, w> \in Q \times (\Sigma \cup \{e\})$ definiert sein.

Deterministische endliche Automaten unterscheiden sich von NEAs dadurch, daß von jedem Zustand unter Verarbeitung eines beliebigen Eingabesymbols höchstens ein Zustand erreichbar ist; d.h. σ ordnet jedem $<q_i, w_j> \in Q \times \Sigma$ ein Element aus Q zu.

Definition (10-2) *Konfiguration*
Wenn M = <Q, Σ, σ, q_0, F> ein endlicher Automat ist, dann ist jedes <q, w> ∈ Q × Σ^* eine *Konfiguration* von M.

Eine Konfiguration <q_0, w> wird als *Anfangskonfiguration* bezeichnet; eine Konfiguration <q, *e*>, mit q ∈ F, als *Endkonfiguration*.

Der Automat M kann von einer Konfiguration <q, aw> in die Konfiguration <q', w> übergehen (notiert als <q, aw> $\vdash$ <q', w>) gdw. q' ∈ σ(q, a)[3].

Definition (10-3) *Akzeptierte Sprache*
Wenn M = <Q, Σ, σ, q_0, F> ein endlicher Automat ist, dann ist die von M *akzeptierte Sprache* definiert als:

$$L(M) = \{w | w \in \Sigma^* \wedge <q_0, w> \vdash^* <q, e>, \text{ für ein beliebiges } q \in F\}.$$

Beispiel (10-1)
M_1 = <{1, 2, 3, 4}, {h, a, !}, σ, 1, {4}>, mit

σ	h	a	!
1	2		
2		3	
3	2		4
4			

M_1 ist ein DEA: Von jedem Zustand ist höchstens ein neuer Zustand erreichbar. Außerdem gibt es Konfigurationen, von denen aus keine andere Konfiguration erreicht werden kann; so z.B. <1, a>. Die von M_1 akzeptierte Sprache besteht aus einer oder mehreren Folgen von "ha"s mit abschließendem "!"; d.h. $L(M_1) = (ha)^+!$.

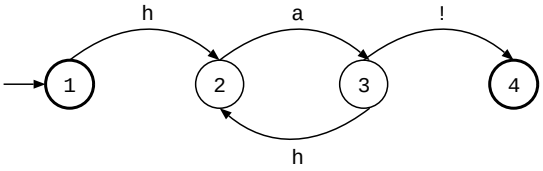
10.2 Repräsentationsmöglichkeiten

Eine beliebte Repräsentationsform für endliche Automaten bilden die endlichen Übergangsnetzwerke (*finite transition networks*): Die Knoten des Netzes repräsentieren die Zustände des Automaten, und die Kanten zwischen zwei Knoten sind mit dem Symbol des Eingabevokabulars etikettiert, das beim Übergang vom ersten zum zweiten Knoten verarbeitet wird. Start- und Endzustände werden besonders markiert.

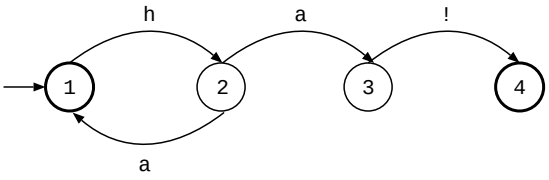
[3] "$\vdash^*$" bezeichnet den reflexiven transitiven Abschluß der Übergangsrelation.

Beispiel (10-2)

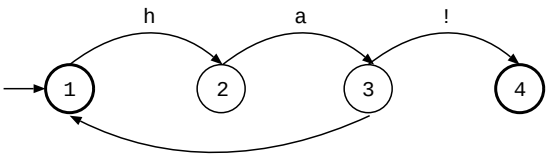
1. Der Automat M_1 wird durch folgendes Übergangsnetzwerk repräsentiert:
   `Netz (1)`



2. Das folgende Übergangsnetzwerk repräsentiert einen NEA, der dieselbe Sprache wie M_1 akzeptiert:
   `Netz (2)`



3. Nicht bei jedem Übergang von einer Konfiguration in eine andere muß ein Symbol der Eingabe verarbeitet werden. In Netzwerken werden Übergänge dieses Typs durch sogenannte "jump-arcs" repräsentiert. Im folgenden Netzwerk sind die Knoten 3 und 1 durch einen *jump-arc* verbunden:
   `Netz (3)`



Es gilt: $L(N_1) = L(N_2) = L(N_3)$. N_1 repräsentiert einen DEA, N_2 und N_3 NEAs.

10.3 Repräsentation endlicher Automaten in LISP

Bei der Implementierung von endlichen Automaten werden wir uns an ihrer Repräsentation als Übergangsnetzwerke orientieren.

Ein Netzwerk wird durch eine Liste repräsentiert, deren ersten beiden Elemente die Start- und Endzustände des Netzwerkes festlegen; die übrigen Elemente spezifizieren die Kanten des Netzwerks.

Als Kantenetiketten werden nicht nur Symbole des Eingabevokabulars zugelassen, sondern auch Kategoriensymbole, die Klassen von Symbolen zusammenfassen. So ist es möglich, alle Kanten (außer *jump*-Kanten), die sich nur durch das Etikett unterscheiden, durch eine Kante zu ersetzen.

Beispiel (10-3)

;;; Ein einfaches Übergangsnetzwerk, das einfache englische Sätze erkennt:

```
(defvar *englisch*)
(setq *englisch*
  '((startknoten (1))
    (endknoten (9))
    (Von 1 nach 3 Etikett NP)          (Von 1 nach 2 Etikett DET)
    (Von 2 nach 3 Etikett N)           (Von 3 nach 4 Etikett BV)
    (Von 4 nach 5 Etikett ADV)         (Von 4 nach 5 Etikett |#|)[4]
    (Von 5 nach 6 Etikett DET)         (Von 5 nach 7 Etikett DET)
    (Von 5 nach 8 Etikett |#|)         (Von 6 nach 6 Etikett MOD)
    (Von 6 nach 7 Etikett ADJ)         (Von 7 nach 9 Etikett N)
    (Von 8 nach 8 Etikett MOD)         (Von 8 nach 9 Etikett ADJ)
    (Von 9 nach 4 Etikett CNJ)         (Von 9 nach 1 Etikett CNJ)))
(defvar *lexikon*)
(setq *lexikon*
  '((NP kim sandy lee)                 (DET a the her)
    (N consumer man woman)             (BV is was)
    (CNJ and or)                       (ADJ happy stupid)
    (MOD very)                         (ADV often always sometimes)))
```

[4] *Jump*-Kanten werden durch das Hash-Symbol "#" markiert. Da dieses Symbol zu den Makro-Zeichen von Common-LISP gehört, muß durch die Verwendung von *Escape*-Zeichen die Interpretation des Hash-Symbols als Makro-Zeichen unterdrückt werden.

Die Struktur des Übergangsnetzwerks *ENGLISCH* verdeutlicht die folgende Abbildung:

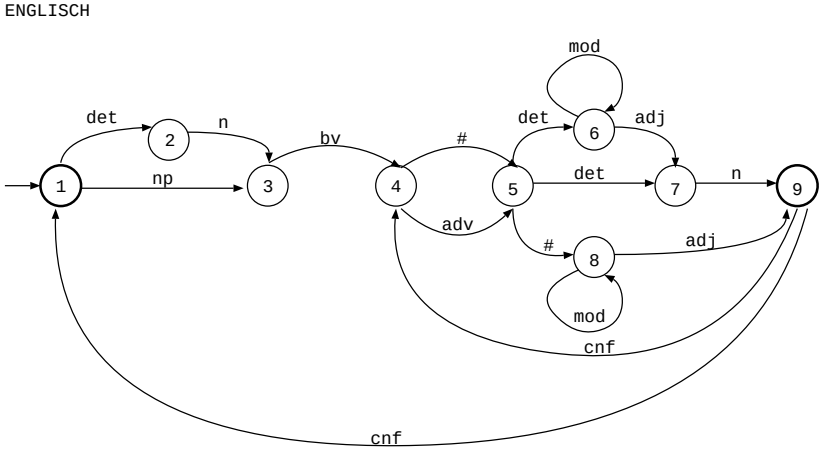


Abbildung (10-1)

Das Netzwerk akzeptiert einfache prädikative Aussagesätze bzw. Konjunktionen solcher Aussagesätze, in denen die Objekt-NP eine beliebige Zahl von Gradadverbien enthalten kann oder aus einer Konjunktion von NPs besteht.

Beispiel (10-4)
Das Übergangsnetzwerk *ENGLISCH* akzeptiert u.a. die folgenden Sätze:

Lee is very ... stupid.

Kim was a consumer and Lee is always happy.

Her man is sometimes her man.

⋮ ⋮ ⋮

Bevor wir einen Algorithmus implementieren, der die Verarbeitung von Symbolfolgen durch ein Netzwerk erlaubt, definieren wir Netzwerk- und Kantenselektoren, die die Transparenz des folgenden Programms fördern.

Selektoren für Netzwerke:

Die folgenden drei Funktionen liefern als Wert die Liste der Start-/Endknoten bzw. aller Kanten eines Netzwerkes als Wert.

```
(defun startknoten (netzwerk)
  (second (assoc 'startknoten netzwerk)))
(defun endknoten (netzwerk)
  (second (assoc 'endknoten netzwerk)))
(defun kanten (netzwerk)
  (nthcdr 2 netzwerk))
```

Selektoren für Kanten:

Die folgenden drei Selektoren erwarten als Argument die Kante eines Netzwerkes. Kanten werden durch Listen der Form (*Startknoten Etikett Endknoten*) repräsentiert. Die Funktionen liefern den Start-/Endknoten bzw. das Etikett der Kante als Wert.

```
(defun knoten1 (kante)
  (second kante))
(defun knoten2 (kante)
  (fourth kante))
(defun etikett (kante)
  (sixth kante))
```

10.4 Recognizer und Generatoren

Eine Symbolfolge (hier repräsentiert durch eine Liste von Symbolen) wird akzeptiert gdw. wenn es im Netzwerk einen Pfad gibt, der unter Verarbeitung der Symbolfolge von einem Startknoten zu einem Endknoten führt. Die Suche terminiert, sobald ein solcher Pfad gefunden wurde oder es keinen weiteren Pfad im Netzwerk mehr gibt, der überprüft werden könnte.

```
;;; RECOGNIZE [Top-Level Funktion]
;;; Argumente: Eingabe - eine Liste von Symbolen und
;;;            Netzwerk - ein Netzwerk
;;; Wert:      T, gdw. Symbolfolge akzeptiert wird; NIL sonst.
(defun recognize (eingabe netzwerk)
  (dolist (initialnode (startknoten netzwerk))[5]
    (when (recognize-next initialnode eingabe netzwerk)
      (return t))))

;;; RECOGNIZE-NEXT [der Motor des Programms]
;;; Argumente: Knoten - der aktuelle Zustand
;;;            Eingabe- die noch zu verarbeitende Symbolfolge
;;;            Netzwerk - das Netzwerk
;;; Wert:      T/NIL
(defun recognize-next (knoten eingabe netzwerk)
  (cond ((and (null eingabe)                          ; Eingabe verarbeitet und
              (member knoten (endknoten netzwerk))) t); Endknoten erreicht.
        ((endp eingabe) nil); in eine Sackgasse geraten
        ;; untersuche alle möglichen Fortsetzungen des begonnen Pfades:
        (t (dolist (kante? (kanten netzwerk))
             (if (and (eq knoten (knoten1 kante?))
                      (recognize-next
                        (knoten2 kante?)
                        (recognize-move (etikett kante?) eingabe)
                        netzwerk))
                 (return t))))))
```

```
;;; RECOGNIZE-MOVE
;;; Hilfsfunktion, die berechnet, wie sich die zu verarbeitende Symbolfolge ändert,
;;; wenn man den Pfad im Netzwerk mit einer bestimmten Kante fortsetzt.
;;; Argumente: Etikett - das Etikett der gewählten Kante
;;;            Eingabe - die noch zu verarbeitende Symbolfolge
;;; Wert :     Die Symbolfolge minus dem ersten Symbol, gdw. das Etikett ein
;;;            Kategoriensymbol ist und das erste Symbol Element dieser Kate-
;;;            gorie ist; die unveränderte Symbolfolge, gdw. das Etikett der Kante
;;;            "#" ist (Jump-Kante); sonst NIL.
(defun recognize-move (etikett eingabe)
  (cond ((or (eq etikett (first eingabe))
             (member (first eingabe) (assoc etikett *lexikon*)))
         (rest eingabe))
        ((eq etikett '|#|) eingabe)
        (t 'stop)))
```

Um zu überprüfen, ob ein endlicher Automat sich so verhält, wie es die Entwickler wünschen, ist es sehr hilfreich, wenn man über ein Programm verfügt, das alle Sequenzen generiert, die der Automat akzeptiert. Ein solches Programm realisiert einen **Generator** für die durch den Automaten festgelegte Sprache.

Das folgende Programm generiert alle Sätze, die ein Übergangsnetzwerk akzeptiert. Eine Einschränkung gilt es zu beachten: Bei Netzwerken, die eine nicht-endliche Anzahl von Symbolfolgen akzeptieren können, wie z.B. unserem Netzwerk *ENGLISCH*, muß sichergestellt werden, daß der Generierungsprozeß nach endlich vielen Schritten terminiert. Zu diesem Zweck wird die Zahl der rekursiven Aufrufe durch die Variable GRENZE beschränkt.

```
;;; GENERATE [Hauptfunktion]
;;; Generiert alle vom Netzwerk erkannten Symbolfolgen, die ohne Überschreitung
;;; der maximalen Rekursionstiefe erzeugt werden können
;;; Argumente:    Netzwerk - ein Netzwerk
;;;               Grenze - die maximale Rekursionstiefe
;;;               Wert: NIL
;;; Seiteneffekt: Ausgabe aller generierten Symbolfolgen
(defun generate (netzwerk grenze)
  (dolist (initialnodes (startknoten netzwerk))
    (generate-next initialnodes nil netzwerk grenze)))
```

[5] Abweichend von Definition (1) lassen wir Netzwerke zu, die mehr als einen Startzustand haben.

```
;;; GENERATE-NEXT [Motor desProgramms]
;;; Argumente: Knoten - der aktuelle Knoten
;;;            Ausgabe - eine Symbolfolge
;;;            Netzwerk - ein Netzwerk
;;;            Grenze - die maximale Rekursionstiefe
;;; Wert & Seiteneffekt: s.o.
(defun generate-next (knoten ausgabe netzwerk grenze)
  (unless (<= grenze 0)
    (if (member knoten (endknoten netzwerk))
        (print ausgabe)
        (dolist (kante? (kanten netzwerk))
          (when (equal knoten (knoten1 kante?))
            (dolist (ausgabe? (generate-move (etikett kante?) ausgabe))
              (generate-next (knoten2 kante?) ausgabe? netzwerk (- grenze 1))))))))

;;; GENERATE-MOVE [Hilfsfunktion]
;;; Argumente: Etikett - das Etikett einer Kante
;;;            Ausgabe - eine Symbolfolge
;;; Wert:      Die Liste aller Symbolfolgen, die sich aus der Symbolfolge durch
;;;            Verkettung mit allen Elementen der durch Etikett bezeichneten
;;;            Kategorie bilden lassen.
(defun generate-move (etikett ausgabe)
  (cond ((eq etikett '|#|) (list ausgabe))
        ((assoc etikett *lexikon*)
         (mapcar #'(lambda (x) (append ausgabe (list x)))
                 (rest (assoc etikett *lexikon*))))
        (t (list (append ausgabe (list etikett))))))
```

10.5 Transducer

Ein Transducer ist ein Automat, der sich von einem einfachen Recognizer darin unterscheidet, daß er für jede akzeptierte Symbolfolge eine '*Übersetzung*' produziert und ausgibt.

Definition (10-4) *Endlicher Transducer*

Ein *endlicher Transducer* M = <Q, Σ, Φ, σ, q_0, F> besteht aus:

1. einer endlichen Menge von Zuständen Q = $\{q_0, q_1, \ldots, q_n\}$;
2. einer endlichen Menge Σ = $\{v_1, \ldots, v_m\}$ von Eingabesymbolen;
3. einer endlichen Menge Φ = $\{w_1, \ldots, w_m\}$ von Ausgabesymbolen
4. der Übergangsfunktion σ:Q × (Σ ∪ $\{e\}$) → Pot(Q × Φ^*);
5. dem Anfangszustand q_0 ∈ Q und
6. einer Menge F (F ist eine Teilmenge von Q) von Endzuständen.

Definition (10-5) *Konfiguration*
Wenn M = <Q, Σ, Φ, σ, q_0, F> ein endlicher Transducer ist, dann jedes <q, v, w> ∈ Q × Σ^* × Φ^* eine *Konfiguration* von M.

Beispiel (10-5)
Der folgende Transducer akzeptiert einfache englische Fragesätze und gibt ihre französische Übersetzung aus:

```
(defvar *englisch-franzoesisch*)
(setq *englisch-franzoesisch*
  '((startknoten (1))
    (endknoten (5))
    (Von 1 nach 2 Etikett WH)
    (Von 2 nach 3 Etikett BE)
    (Von 3 nach 4 Etikett DET)
    (Von 4 nach 5 Etikett NOUN)))
(defvar *lexikon*)
(setq *lexikon*
  '((WH (where ou))
    (BE (is est))
    (DET (the |#|))
    (NOUN (exit la_sortie) (policeman le_gendarme) (shop la_boutique)
          (toilet la_toilette))))
```

Das folgende Programm ermöglicht es, Transducer dieses Typs zur Übersetzung von Symbolfolgen zu verwenden. Der entscheidende Unterschied gegenüber der Implementierung des Recognizers am Anfang des Kapitels liegt darin, daß das zweite Argument der Funktionen TRANSDUCE-NEXT und TRANSDUCE-MOVE eine Liste ist, die zwei Listen enthält: die Liste der noch zu verarbeitenden Symbole und die Liste mit der Übersetzung der schon verarbeiteten Symbole.

```
;;; TRANSDUCE [Hauptfunktion]
;;; Sucht nach einem Pfad im Netzwerk, der eine vollständige Verarbeitung der
;;; Eingabe erlaubt.
;;; Argumente: Eingabe - eine Liste von Symbolen und
;;;            Netzwerk - ein Netzwerk
;;; Werte:     eine Übersetzung der Symbolfolge, gdw. sie verarbeitet werden konnte;
;;;            NIL sonst.
(defun transduce (eingabe netzwerk)
  (dolist (s-knoten (startknoten netzwerk))
    (return-if (transduce-next s-knoten (list eingabe ()) netzwerk))))

;;; TRANSDUCE-NEXT [Motor des Programms]
;;; Argumente: Knoten - der aktuelle Knoten;
;;;            ein-aus - eine Liste, die die Liste der noch zu verarbeitenden Symbole
;;;                      und die Liste mit der Übersetzung der schon verarbeiteten
;;;                      Symbole enhält und
;;;            Netzwerk - ein Netzwerk
;;; Wert:      die vollständige Übersetzung der ursprünglichen Symbolfolge, gdw.
;;;            sie verarbeitet werden konnte; NIL sonst.
(defun transduce-next (knoten ein-aus netzwerk)
  (cond ((stringp ein-aus) (format t "~A%" ein-aus))
        ((and (endp (first ein-aus)) (member knoten (endknoten netzwerk)))
         (second ein-aus))
        ((endp (first ein-aus)) nil)
        (t (dolist (kante? (kanten netzwerk))
             (return-if (and (eq knoten (knoten1 kante?))
                             (transduce-next
                               (knoten2 kante?)
                               (transduce-move (etikett kante?) ein-aus)
                               netzwerk)))))))
```

Die beiden voranstehenden Funktionen verwenden einen selbstdefiniertes Makro RETURN-IF, das den Wert seines Arguments mit RETURN zurückgibt, wenn es nicht zu NIL evaluiert.[6]

[6]

```
;;; TRANSDUCE-MOVE
;;; Hilfsfunktion, die die Konsequenz eines Zustandswechsels berechnet
;;; Argumente: etikett - das Etikett eines Knotens
;;;            ein-aus - die Liste, die den Rest der Eingabe und die
;;;                      Übersetzung des Anfangs der Eingabe enthält
;;; Werte:     ([Eingabe - 1. Symbol] [Übersetzung+Übersetzung 1. Symbol]), gdw.
;;;            das 1. Symbol ein Element der durch "Etikett" bezeichneten Kategorie
;;;            ist; sonst NIL.
(defun transduce-move (etikett ein-aus)
  (let ((eintrag nil))
    (cond ((eq etikett '|#|) ein-aus)
          ((setq eintrag
                 (assoc (caar ein-aus) (rest (assoc etikett *lexikon*))))
           (if (eq (second eintrag) '|#|)
               (list (cdar ein-aus) (second ein-aus))
               (list (cdar ein-aus)
                     (append (second ein-aus) (list (second eintrag))))))
          (t "Fehlerhafte lexikalische Analyse!"))))
```

```
(defmacro return-if (argument)
  `(let ((wert ,argument))
     (if wert (return wert))))
```

Kapitel 11

Makros

11.1 Begriffsklärung

Makros bzw. Makro-Funktionen unterscheiden sich von normalen Funktionen darin, daß die Evaluierung einer Form, die einen Makroaufruf enthält, nicht wie im Falle eines Funktionsaufrufs direkt zur Berechnung eines Wert führt, sondern sich in zwei Schritten vollzieht: Zunächst wird die Makroform zu einer LISP-Form expandiert, die dann anschließend evaluiert wird.

1. Durch die Evaluierung eines Makroaufrufs wird eine neue Form, *Zwischenform* genannt, erzeugt. Bei der Generierung der Zwischenform werden die Argumente des Makros nicht evaluiert.

2. Die Zwischenform wird evaluiert und der daraus resultierende Wert als Wert des Makroaufrufs zurückgegeben.

$$\text{Makroaufruf} \longrightarrow \text{Zwischenform} \longrightarrow \text{Wert der Zwischenform}$$

Aufgrund dieser Charakteristik erleichtern Makros die Entwicklung von Funktionen und Programmen, die sich für die Benutzer durch einfache Syntax und ein hohes Maß an Transparenz auszeichnen: Die Expansion von Makroaufrufen während der Ausführung oder Compilierung eines Programms verbirgt die tatsächliche Komplexität des Kodes vor den Augen der Benutzer.

11.2 Motivation

Ein Bereich, der sich besonders gut eignet, die Nützlichkeit und z.T. Notwendigkeit von Makro-Funktionen zu demonstrieren, bildet die Realisierung benutzerdefinierter Kontrollstrukturen.

Nehmen wir an, wir würden mit einem karg ausgestatteten LISP-System arbeiten (ein abgespecktes XLISP z.B.), in dem es als Kontrollstruktur nur die vordefinierte Makrodefinition COND gäbe.

Um in gewohnter Weise weiterarbeiten zu können, wäre nichts naheliegender als der Versuch, sich die fehlenden Strukturen wie IF, WHEN, UNLESS, etc. selbst zu definieren. Wir wissen, daß gilt:

$$(\text{if test } \alpha\ \beta) \equiv (\text{cond (test } \alpha) \text{ (t } \beta))$$

Also scheint nichts dagegen zu sprechen, IF als einfache Funktion zu realisieren:

Beispiel (11-1)

```
> (defun our-if (test then else)
    (cond (test then)
          (t else)))
  OUR-IF
> (our-if t 1 0)
  1
> (our-if nil 1 0)
  0
```

Die Erinnerung an die Tatsache, daß eine Funktion im Gegensatz zu einem Makro immer ihre Argumente evaluiert, läßt schnell Zweifel an der Adäquatheit unserer Realisierung von OUR-IF aufkommen: Es gibt mindestens zwei Fälle, in denen die Evaluierung der Argumente zu Problemen führt:

1. Wenn die Ausführung der *Then*- oder *Else*-Form mit Seiteneffekten verbunden ist: In diesem Fall werden die Seiteneffekte erzielt, unabhängig davon, ob die *Test*-Form zu NIL evaluiert oder nicht.

```
> (defun teste-wert (variable)
    (our-if (< 0 variable)
            (princ "Positiver Wert")
            (princ "Negativer Wert")))
  TESTE-WERT
> (teste-wert 7)
  Positiver WertNegativer Wert
  "Positiver Wert"
> (teste-wert -6)
  Positiver WertNegativer Wert
  "Negativer Wert"
```

2. Es gibt Fälle, in denen die Evaluierung der *Then*- oder *Else*-Form zu einem Fehler führt, sofern nicht die Bedingung erfüllt ist.

```
;; A sei ein Symbol, dem bislang kein Wert zugewiesen wurde:
> (our-if (boundp 'a) a nil)
  "Error: unbound variable A"
  1>
```

Natürlich könnte man die Evaluierung der Argumente durch QUOTE unterdrücken und abhängig von der Evaluierung der *Test*-Form durch EVAL eine zusätzliche Evaluierung der *Then*- bzw. *Else*-Form erzwingen. Das Ergebnis wäre weder benutzerfreundlich (in einigen Kontrollstrukturen sind Argumente zu *quoten*, in anderen dagegen nicht), geschweige denn effizienter und transparenter Kode.

Probleme dieser Art lassen sich vermeiden, indem Funktionen wie OUR-IF als Makro-Funktionen realisiert werden. Zur Definition von Makros dient das DEFMACRO-Makro:

DEFMACRO *Name Lambda-Liste* {*Form*}* [Makro]
DEFMACRO ist eine makrodefinierende Makro-Funktion. Die Syntax ist mit der von DEFUN identisch. Durch die Evaluierung einer DEFMACRO-Form wird dem Symbol *Name* als Funktionsbindung die in der DEFMACRO-Form spezifizierte Makro-Funktion zugewiesen.

Genauso wie man mit SYMBOL-FUNCTION auf die globale Funktionsdefinition eines Symbols zugreifen kann, kann man mit MACRO-FUNCTION auf die globale Makrodefinition[1] eines Symbols zugreifen:

MACRO-FUNCTION *Symbol* [Funktion]
Wenn die Funktionsbindung des Symbols eine Makrodefinition ist, wird die Expansionsfunktion zurückgegeben; sonst NIL.

Ein Symbol kann entweder eine Funktion oder einen Makro bezeichnen, nicht aber beides.

Beispiel (11-2)

```
> (defmacro our-if (test then else)
    (list 'cond (list test then)
          (list t else)))
  OUR-IF
```

[1] Neben globalen Makrodefinitionen können mit dem MACROLET Macro auch lokale Makrodefinitionen generiert werden (vgl. FLET).

```
> (teste-wert 7)
  "Positiver Wert"
> (teste-wert -6)
  ¨Negativer Wert¨
> (our-if (boundp 'a) a 'ungebundenes_Atom)
  UNGEBUNDENES_ATOM

;;; Die Makro-Funktion IF+ unterscheidet sich von der vordefinierten IF-Funktion darin,
;;; daß als Then-/Else-Form nicht eine einzelne Form, sondern eine Liste von Formen
;;; verwendet wird.
> (defmacro if+ (test then else)
    (list 'cond (cons test then)
          (cons t else)))
  IF+
> (defun zahlenspiel (expr)
    (if+ (numberp expr)
         ((print expr)
          (print (* expr expr))
          (* expr expr expr))
         (nil)))
  ZAHLENSPIEL
> (zahlenspiel 3)
  3
  9
  27
```

Die Fehlersuche bei der Entwicklung von Makro-Funktionen wird erheblich vereinfacht, wenn man in der Lage ist zu überprüfen, welche *Zwischenform* bei der Evaluierung eines Makroaufrufes erzeugt wird. In Common LISP gibt es zu diesem Zweck die beiden Funktionen MACROEXPAND-1 und MACROEXPAND:

MACROEXPAND-1	*Form*	[Funktion]
Wenn *Form* ein Makroaufruf ist, dann wird *Form* einmal expandiert und es werden zwei Werte zurückgegeben: die erzeugte (Zwischen)Form und T. Wenn *Form* kein Makroaufruf ist, liefert MACROEXPAND-1 *Form* und NIL als Werte zurück.		

MACROEXPAND	*Form*	[Funktion]
MACROEXPAND unterscheidet sich von MACROEXPAND-1 nur darin, daß *Form* solange expandiert wird, bis das Ergebnis kein Makroaufruf mehr ist.		

Beispiel (11-3)

```
> (macroexpand-1 '(our-if (boundp 'a) a 'ungebundenes_Atom))
  (COND ((BOUNDP 'A) A) (T 'UNGEBUNDENES_ATOM)))
  T
> (macroexpand-1 '(if+ (numberp 3) ((print 3) (print (* 3 3)) (* 3 3 3)) (nil)))
  (COND ((NUMBERP 3) (PRINT 3) (PRINT (* 3 3)) (* 3 3 3)) (T NIL))
  T
> (macroexpand-1 '(teste-wert 7))
  (TESTE-WERT 7)
  NIL

;;; Das folgende Beispiel läßt die Unterschiede von MACROEXPAND-1 und MACRO-
;;; EXPAND deutlich werden:
;;; Wir definieren mit Hilfe des zuvor definierten Makros OUR-IF einen weiteren Makro
;;; OUR-WHEN, der im Unterschied zum vordefinierten WHEN-Makro als zweites Argu-
;;; ment nur eine einzelne Form akzeptiert.
> (defmacro our-when (test aktion)
    (list 'our-if test aktion nil))
  OUR-WHEN
> (setf liste '(a b c))
  (A B C)
> (our-when (listp liste) (first liste))
  A
> (macroexpand-1 '(our-when (listp liste) (first liste)))
  (OUR-IF (LISTP LISTE) (FIRST LISTE) NIL)
> (macroexpand '(our-when (listp liste) (first liste)))
  (COND ((LISTP LISTE) (FIRST LISTE)) (T NIL))
```

11.3 Backquote, Komma, Komma-@

Die Spezifizierung der Zwischenformen über Verwendung von Konstruktoren wie LIST, CONS, etc. ist relativ umständlich und führt zu Kode von geringer Transparenz. Der in LISP zur Verfügung stehende *Backquote-Mechanismus* erleichtert die Erstellung von gut lesbaren Makrodefinitionen erheblich.

Das Backquote ‘ wirkt prinzipiell genauso wie ein einfaches Quote; d.h. es verhindert die Evaluierung des folgenden Ausdrucks. Es gibt allerdings eine Ausnahme: Alle innerhalb einer Backquote-Form vorkommenden Teilausdrücke, die direkt nach einem Komma stehen, werden evaluiert:

Beispiel (11-4)

```
> (setf sym 'beispiel)
  BEISPIEL
> `(nur ein kleines ,sym)
  (NUR EIN KLEINES BEISPIEL)

;;; Mit Hilfe des Backquote-Mechanismus lassen sich die Makros OUR-IF und
;;; OUR-WHEN wie folgt definieren:
> (defmacro our-if (test then else)
    `(cond (,test ,then)
          (t ,else)))
OUR-IF
> (defmacro our-when (test aktion)
    `(our-if ,test ,aktion nil))
  OUR-WHEN
```

Neben dem Komma kann innerhalb einer Backquote-Form die Kombination Komma-At [,@] verwendet werden. Der Unterschied zu einem einfachen Komma liegt darin, daß der folgende Ausdruck zu einer Liste evaluieren muß, die dann ohne ihre äußeren Klammern in die Backquote-Form eingefügt wird:

Beispiel (11-5)

```
> (setf fruechte '(apfel banane kirsche))
  (APFEL BANANE KIRSCHE)
> `(ich esse gerne: ,@fruechte)
  (ICH ESSE GERNE: APFEL BANANE KIRSCHE)

;;; Reformulierung von IF+:
> (defmacro if+ (test then else)
    `(cond (,test ,@then)
          (t ,@else)))
  IF+
```

Ein weiteres Beispiel zeigt die Bedeutung von Makros für die Realisierung angenehmer Arbeitsbedingungen: Die Editor-Einbindung in unserem XLISP-System ist durch eine Funktion realisiert, die als optionales Argument einen String erwartet. Die in anderen Systemen vordefinierte Funktion ED ist normalerweise als Makro realisiert, so daß als Argument ein (nicht gequotetes) Symbol verwendet werden kann. Als Makro könnte ED z.B. folgendermaßen definiert werden:

Beispiel (11-6)

```
;; der Name der zuletzt geladenen Datei wird ein einer globalen Variable gespeichert:
(defvar *aktueller-file*)

(defmacro ed (&optional²file)
  `(cond ((and (null ',file) *aktueller-file*)
          ;; kein Dateiname angegeben, aber *AKTUELLER-FILE* hat einen
          ;; Wert ungleich NIL ⟹ Aufruf von ED mit diesem Wert
          (ed ,*aktueller-file*))
         ((null ',file)
          ;; kein Dateiname angegeben, und *AKTUELLER-FILE* = NIL
          ;; ⟹ Benutzer auffordern, einen Dateinamen anzugeben:
          (princ "Name der zu edierenden Datei: ")
          (setf *aktueller-file* (symbol-name (read)))
          (ed ,*aktueller-file*))
         ((symbolp ',file)
          ;; Dateiname angegeben und Dateiname ist ein Symbol:
          ;; wandle Symbol in String um und rufe ED rekursiv mit ihm auf:
```

```
 (ed (symbol-name ',file)))
((if (and (> (length ,file) 4)
          (equal (subseq ,file (- (length ,file) 4) (length ,file)) ".lsp"))
    (setq *aktueller-file* ,file)
    (setq *aktueller-file* (strcat ,file ".lsp")))
 ;; Dateiname angegeben und Dateiname ist ein String
 ;; ⟹ 1. a) Prüfe die Extension des Dateinames und
 ;; b) Aktualisiere *AKTUELLER-FILE*
 ;; 2. Rufe Editor mit Datei auf
 ;; 3. Lade die (modifizierte) Datei
 ;; 4. Lösche den Bildschirm
 (system (strcat "jove " *aktueller-file*))
 (load *aktueller-file*)
 (system "cls"))))
```

11.4 Destrukturierung

Die Möglichkeit, mit Makros Konstrukte zu entwickeln, deren Syntax sich von der üblichen LISP-Syntax abhebt, basiert auf einer Eigenschaft von Makros, die als "destructuring" bezeichnet wird. Zwei Destrukturierungsregeln sind zu unterscheiden:

1. An jeder Position in der Lambda-Liste einer Makrodefinition kann statt eines Symbols eine Lambda-Liste spezifiziert werden. Die Elemente dieser Liste werden beim Makroaufruf in geeigneter Weise an die beim M akroaufruf verwendeten Objekte gebunden.

2. Die Lambda-Liste darf als *dotted-pair* behandelt werden, die mit einem Parameternamen endet. In diesem Fall wird der Parameter an die Liste aller beim Makroaufruf noch nicht gebundenen Argumente gebunden (vgl. &rest-Schlüsselwort, Kap.12).

Beispiel (11-7)

```
;;; Die folgende, nicht besonders sinnvolle Definition illustriert die Konsequenzen der
;;; ersten Regel:
> (defmacro unsinn ((zahl1 . zahlen) operator)
    `(,operator ,zahl1 ,@zahlen))
  UNSINN
```

[2] "&optional" ist ein sogenanntes *'lambda-list keyword'*. Alle folgenden Parameter sind optional. Innerhalb eines Funktionaufrufs nicht spezifizierte optionale Parameter werden auf den *default*-Wert NIL gesetzt.

```
> (unsinn (1 2 3 4 5) +)
  ;; ZAHL1 <= 1
  ;; ZAHLEN <= (2 3 4 5)
  ;; OPERATOR <= +
  15

;;; Die zweite Regel nutzen wir aus, um das LET-Makro zu reimplementieren:

;;; 1. Variante: (ohne Backquote)
> (defmacro let (variablen-&-werte . body)
    (cons (cons 'lambda
                (cons (mapcar #'let-variablen variablen-&-werte) body))
          (mapcar #'let-werte variablen-&-werte)))
  LET
> (defun let-variablen (item)
    (if (listp item)
        (first item)
        item))
    LET-VARIABLEN
> (defun let-werte (item)
    (if (listp item)
        (second item)
        nil))
    LET-WERTE

;;; Wie man sieht, wird die LET-Form in eine Lambda-Form überführt:
> (macroexpand '(let ((a 9) (b 7) c) (+ a b)))
    ((LAMBDA (A B C) (+ A B)) 9 7 NIL)

;;; 2. Variante: (mit Backquote)
> (defmacro let (variablen-&-werte . body)
      `((lambda ,(mapcar #'let-variablen variablen-&-werte) ,@body)
        ,@(mapcar #'let-werte variablen-&-werte)))
```

Kapitel 12

Lambda-Listen Schlüsselwörter

Wir werden in diesem Kapitel die vollständige Syntax von DEFUN- bzw. LAMBDA-Formen vorstellen und verschiedene Typen von Parametern einführen, die es erlauben Funktionen zu definieren, deren Lambda-Listen neben *normalen* obligatorischen Parametern u.a. auch optionale und *keyword*-Parameter enthält. Durch die Verwendung solcher Parameter lassen sich Funktionen entwickeln, die sich durch ein großes Maß an Flexibilität auszeichnen. Allerdings besteht bei exessiver Verwendung dieser Sprachmittel die Gefahr, zunehmend prozedurale Aspekte in die Lambda-Liste zu verlagern und so Kode von geringer Transparenz zu erzeugen.

Bislang haben wir bei der Definition von Funktionen nur einen geringen Teil der Möglichkeiten ausgeschöpft, die LAMBDA- & DEFUN-Formen bieten. Der folgende reguläre Ausdruck beschreibt die vollständige Syntax dieser Formen:

```
1  (lambda ({var}*
2              [ &optional { var | (var [initform [svar]])}*]
3              [ &rest var]
4              [ &key { var | ({var | (keyword var)} [initform [svar]])}*]
5              [ &aux { var | (var [initform])}*]
6    {Deklaration | Dokumentations-String}*
7    {form}*)
```

Die Lambda-Liste einer LAMBDA- bzw. DEFUN-Form besteht aus einer Folge (obligatorischer) Variablen, gefolgt von einer Zahl von *Lambda-Listendeklarationen* (2-5). Eine Lambda-Listendeklaration besteht aus einem Lambda-Listenschlüsselwort, gefolgt von einer oder mehreren *Variablendeklarationen*. Eine Variablendeklaration besteht aus einem Variablennamen, auf den optional noch eine Initialisierungsform folgen kann. Es gilt: *var* und *svar* stehen für beliebige Symbole, und *initform* kann eine beliebige LISP-Form sein.

Die oben spezifizierte Reihenfolge für die Lambda-Listenschlüsselwörter (*&optional* vor *&rest* vor *&key* vor *&aux*) ist obligatorisch.

12.1 Syntax der Parameterliste

Für die Lambda-Listenschlüsselwörter (LLSs) gelten die folgenden syntaktischen Regeln:

- *&optional*
  Die auf dieses Schlüsselwort folgenden Parameter (bis zum nächsten LLS oder dem Ende der Lambda-Liste) werden als optionale Parameter behandelt.

- *&rest*
  Es darf nur ein *&rest*-Parameter spezifiziert werden, auf den ein anderes LLS folgt oder der die Lambda-Liste abschließt.

- *&key*
  Die folgenden Parameter (bis zum nächsten LLS oder dem Ende der Lambda-Liste) werden als *keyword*-Parameter interpretiert.

- *&aux*
  Alle folgenden Parameter bezeichnen lokale Variablen ("**aux**iliary variables"). Auf das *&aux*-Schlüsselwort darf kein anderes Schlüsselwort folgen.

12.2 Semantik der Parameterliste

Bei einem Funktionsaufruf werden die Argumente von links nach rechts evaluiert und die so berechneten Werte an die Parameter der Funktion gebunden. Wenn es ausschließlich obligatorische Parameter gibt, muß die Zahl der Argumente der Zahl der Parameter entsprechen; anderenfalls muß die Zahl der Argumente mindestens so groß sein, wie die Zahl der obligatorischen Argumente.

Für alle nicht-obligatorischen Parameter gelten die folgenden Regeln:

1. Wenn optionale Parameter angegeben sind, dann wird jeder von ihnen wie folgt verarbeitet:

   - Gibt es noch zu verarbeitende Argumente, wird dem Parameter der Wert des ersten nicht verarbeiteten Arguments zugewiesen.
   - Anderenfalls wird ihm der Wert der *initform* zugewiesen bzw. NIL, falls keine *initform* spezifiziert ist.

- Wurde für den Parameter neben der *initform* auch eine Variable *svar* angegeben, dann wird ihr T zugewiesen, sofern dem Parameter der Wert eines Arguments zugewiesen wurde; sonst NIL.

2. Wenn ein *rest*-Parameter spezifiert ist, dann wird ihm als Wert die Liste aller noch nicht verarbeiteten Argumentwerte des Funktionsaufrufs zugeordnet.

3. Wenn in der Lambda-Liste *key*-Parameter spezifiziert sind, muß im Funktionsaufruf nach Verarbeitung der optionalen und des *rest*-Parameters noch eine gerade Zahl von nicht verarbeiteten Argumenten übrig sein, die als eine Folge von Paaren betrachtet wird:

   - Das erste Argument jedes Paares wird als *keyword*-Name interpretiert[1] und das zweite als Bezeichner des korrespondierenden Wertes.
   - In jedem Bezeichner eines *keyword*-Parameters muß ein Symbol enthalten sein, das den Parameter benennt. Ist darüberhinaus explizit ein *keyword*-Name spezifiziert, wird er als *keyword*-Bezeichner verwendet. Anderenfalls ergibt sich der Bezeichner aus dem Parameternamen durch Voranstellung eines Doppelpunktes.
   - *Keyword*-Parameter werden von links nach rechts verarbeitet: Gibt es im Funktionsaufruf ein Paar, dessen *keyword*-Name mit dem des *keyword*-Bezeichners der Lambda-Liste übereinstimmt, wird der entsprechende Parameter an den Wert des zweiten Elementes des Paares gebunden; sonst wird ihm NIL bzw. der Wert der *initform* zugewiesen.
   - Die Behandlung einer weiteren, im *keyword*-Parameterbezeichner enthaltenen Variable (svar), erfolgt wie bei optionalen Parametern.

4. Wenn sowohl ein *rest*-Parameter als auch *keyword*-Parameter spezifiziert sind, werden dieselben Argumente für beide Parametertypen verarbeitet.

5. Enthält die Lambda-Liste noch *aux*-Parameter, werden sie von links nach rechts verarbeitet und an die Werte der übrigen Argumente bzw. des Werts der *initform* oder NIL gebunden.

Beispiel (12-1)

```
;;; (1) &optional und &rest-Parameter[2]:
> ((lambda (a b) (+ a (* b 3))) 4 5)
  19
```

[1] Schlüsselwörter ("keywords") beginnen in Common LISP immer mit einem Doppelpunkt und werden als Konstanten interpretiert, die zu sich selbst evaluieren.

```
> ((lambda (a &optional (b 2)) (+ a (* b 3))) 4 5)
;; Da genug Argumente spezifiziert wurden, um alle Parameter der Funk-
;; tion zu binden, wird die Init-Form für B nicht evaluiert:
;; A ⟸ 4
;; B ⟸ 5
   19
> ((lambda (a &optional (b 2)) (+ a (* b 3))) 4)
;; Da jetzt B an keines der Argumente gebunden werden kann, wird die B
;; zugeordnete Init-Form evaluiert:
;; A ⟸ 4
;; B ⟸ 2
   10
> ((lambda (&optional (a 2 b) (c 3 d) &rest x) (list a b c d x)))
;; B, D sind die svar-Variablen für A und C.
;; Da die Form ohne Argumente aufgerufen wird, gilt:
;; A ⟸ 2
;; C ⟸ 3
;; B, D ⟸ NIL
;; X ⟸ NIL
   (2 NIL 3 NIL NIL)
> ((lambda (&optional (a 2 b) (c 3 d) &rest x) (list a b c d x)) 6)
;; Da A an ein Argument gebunden werden kann, gilt:
;; B ⟸ T
   (6 T 3 NIL NIL)
> ((lambda (&optional (a 2 b) (c 3 d) &rest x) (list a b c d x)) 6 3)
   (6 T 3 T NIL)
> ((lambda (&optional (a 2 b) (c 3 d) &rest x) (list a b c d x)) 6 3 8)
;; Da das Argument 8 nicht an einen Parameter gebunden werden kann,
;; gilt: X ⟸ (8)
   (6 T 3 T (8))
> ((lambda (&optional (a 2 b) (c 3 d) &rest x) (list a b c d x)) 3 6 8 9 10 11)
   (6 T 3 T (8 9 10 11))

;;; (2) &key-Parameter:

> ((lambda (a b &key c d) (list a b c d)) 1 2)
;; Wenn key-Parameter nicht gebunden werden, evaluieren sie zu NIL:
   (1 2 NIL NIL)
> ((lambda (a b &key c d) (list a b c d)) 1 2 :c 6)
```

```
;; Durch ":c 6" wird dem &key-Parameter C der Wert 6 zugewiesen. Damit
;; sind alle Argumente verbraucht und D evaluiert zu NIL.
  (1 2 6 NIL)
> ((lambda (a b &key c d) (list a b c d)) 1 2 :d 8)
  (1 2 NIL 8)
> ((lambda (a b &key c d) (list a b c d)) 1 2 :c 6 :d 8)
  (1 2 6 8)
> ((lambda (a b &key c d) (list a b c d)) 1 2 :d 8 :c 6)
  (1 2 6 8)
> ((lambda (a b &key c d) (list a b c d)) :a 1 :d 8 :c 6)
  ;; Die Variable A wird an das keyword :a gebunden.
  (:a 1 6 8)
> ((lambda (a b &key c d) (list a b c d)) :a :b :c :d)
  (:a :b :d NIL)

;;; (3) Vermischtes:
> ((lambda (a &optional (b 3) &rest x &key c (d a)) (list a b c d x)) 1)
;; Dieses Beispiel enthält &optional-, &rest- und &key-Parameter:
;; B ⟸ (Wert der Init-Form)
;; X ⟸ NIL (keine unverarbeiteten Variablen)
;; D ⟸ 1 (Initform liefert Wert von A)
  (1 3 NIL 1 NIL)
> ((lambda (a &optional (b 3) &rest x &key c (d a)) (list a b c d x)) 1 2)
  (1 2 NIL 1 NIL)
> ((lambda (a &optional (b 3) &rest x &key c (d a)) (list a b c d x)) :c 7)
  (:c 7 NIL :c NIL)
> ((lambda (a &optional (b 3) &rest x &key c (d a)) (list a b c d x)) 1 6 :c 7)
  (1 6 7 1 (:c 7))
> ((lambda (a &optional (b 3) &rest x &key c (d a)) (list a b c d x)) 1 6 :d 8)
  (1 6 NIL 8 (:d 8))
> ((lambda (a &optional (b 3) &rest x &key c (d a)) (list a b c d x)) 1 6 :d 8 :c 9 :d 10)
  (1 6 9 8 (:d 8 :c 9 :d 10))
```

[2] Übernommen aus Steele(1984), S.63-65.

Mit Hilfe der Lambda-Listenschlüsselwörter läßt sich eine LISP-Implementierung, die sich auf die Kernfunktion von Common LISP beschränkt, ohne viel Aufwand so erweitern, daß fast die Funktionalität einer Vollimplementation erreicht wird. Eine gutes Beispiel bietet die *public domain* Implemtierung XLISP 2.1. Viele Common LISP Funktionen werden erst in der Initialisierungsdatei (*init.lsp*) definiert:

Beispiel (12-2)

```
;;; BUTLAST
;;; Durch den optionalen Parameter n wird die Zahl der Elemente spezifiziert, die am Li-
;;; stenende entfernt werden sollen. Wird beim Funktionsaufruf kein Argument angegeben,
;;; an dessen Wert n gebunden werden kann, wird n der default-Wert 1 zugewiesen.
(defmacro butlast (liste &optional n)
  `(if ,n (remove-last ,liste ,n) (remove-last ,liste 1)))

;;; REMOVE-LAST
;;; Entfernt die N letzten Elemente von LISTE.
(defun remove-last (liste n)
  (let ((elemente (length liste)))
    (if (>= n elemente)
        ()
        (do ((in liste (rest in))
             (zaehler (- elemente n) (- zaehler 1))
             (resultat () (append resultat (list (first in)))))
            ((= zaehler 0) resultat)))))

;;; WITH-OPEN-FILE
;;; Der rest-Parameter ermöglicht es, eine beliebige Zahl von Formen innerhalb einer
;;; WITH-OPEN-FILE Form anzugeben, die nacheinander evaluiert werden (PROGN).
;;; Die UNWIND-PROTECT Funktion stellt sicher, daß die Anweisung, die ihr zweites
;;; Argument bildet, auch dann ausgeführt wird, wenn es bei der Evaluierung ihres ersten
;;; Arguments zu einem Fehler kommt. In diesem Fall wird also die Datei auf jeden Fall
;;; geschlossen, auch wenn die Ausführung des Anweisungsblocks der WITH-OPEN-FILE
;;; Form einen Fehler verursacht.
(defmacro with-open-file (angaben &rest body)
  `(let ((,(first angaben)
          (open ,(second angaben) :direction ,(fourth angaben))))
     (unwind-protect
       ,(cons 'progn body)
       ,(list 'close (first angaben)))))
```

```
;;; RASSOC
;;; Die Verwendung des keyword-Parameters test mit dem default-Wert #'eql gestattet es,
;;; RASSOC bei der Suche nach einem Element einer Assoziationsliste, dessen CDR mit
;;; dem Argument wert übereinstimmt, mit einer anderen Vergleichsfunktion als EQL zu
;;; verwenden.
(defun rassoc (wert a-list &key (test #'eql))
  (dolist (x a-list)
    (when (funcall test wert (rest x)) (return x))))
```

```
> (rassoc '4 '((a . 1) (b . 2) (c . 3) (d . 4)))
  (d . 4)
> (rassoc '(4) '((a 1) (b 2) (c 3) (d 4)))
  NIL
> (rassoc '(4) '((a 1) (b 2) (c 3) (d 4)) :test #'equal)
  (D 4)
```

Es gibt eine ganze Reihe von Funktionen in Common LISP, die die Angabe eines *test*-keyword Argumentes erlauben, durch das die von der Funktion zu verwendende Vergleichsfunktion spezifiziert werden kann. So z.B ASSOC, MEMBER, SUBST und REMOVE:

Beispiel (12-3)

```
> (assoc 'c '((a 1) (b 2) (c 3) (d 4)))
  (C 1)
> (assoc '(c) '(((a) 1) ((b) 2) ((c) 3) ((d) 4)))
  NIL
> (assoc '(c) '(((a) 1) ((b) 2) ((c) 3) ((d) 4)) :test #'equal)
  ((C) 3)
> (member 'c '(a b c d e))
  (C D E)
> (member '(c) '((a) (b) (c) (d) (e)))
  NIL
> (member '(c) '((a) (b) (c) (d) (e)) :test #'equal)
  ((C) (D) (E))
> (remove 'a (a b a c a d a))
  (B C D)
> (remove '(a) ((a) b (a) c (a) d (a)))
  ((A) B (A) C (A) D (A))
> (remove '(a) ((a) b (a) c (a) d (a)) :test #'equal)
```

(B C D)

Kapitel 13

Zeichen & Zeichenketten

13.1 Zeichen

Die Behandlung von Zeichen in Common-LISP ist nicht unproblematisch. Wir beschränken uns in diesem Zusammenhang nur auf die wichtigsten Zeichen-Funktionen.

13.1.1 Zeichenprädikate

Die Bedeutung der Prädikate UPPER-CASE-P, LOWER-CASE-P und DIGIT-CHAR-P ist offensichtlich; BOTH-CASE-P trifft auf die Zeichen zu, für die es sowohl eine upper-case wie lower-case Variante gibt. An Vergleichsprädikaten gibt es: CHAR=, CHAR/=, CHAR<, CHAR>, CHAR<= und CHAR>= [1].

Beispiel (13-1)

```
> (lower-case-p #\a)
  T
> (upper-case-p #\a)
  NIL
> (both-case-p #\a)
  T
> (both-case-p #\0)
  NIL
> (both-case-p #\space)
  NIL
```

[1] Die diesen Funktionen korrespondierenden Funktionen, die keine Unterschiede zwischen Groß- und Kleinbuchstaben machen, lauten: CHAR-EQUAL, CHAR-NOT-EQUAL, CHAR-LESSP, CHAR-GREATERP, CHAR-NOT-GREATERP und CHAR-NOT-LESSP.

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

13.1.2 Zeichengenerierung und -konvertierung

Zur Zeichen-Kode Konvertierung dienen die Funktionen CHAR-CODE und CODE-CHAR: CHAR-CODE akzeptiert ein Zeichen als Argument und liefert dessen Kode (nicht-negative ganze Zahl); CODE-CHAR generiert das Zeichen, das dem als Argument übergebenen Kode entspricht.

Die Funktionen CHAR-UPCASE und CHAR-DOWNCASE erzeugen (sofern möglich) aus einem Kleinbuchstaben einen Großbuchstaben bzw. aus einem Großbuchstaben einen Kleinbuchstaben.

Beispiel (13-2)

```
> (char-code #\space)
  32
> (char-code #\9)
  57
> (code-char 88)
  #\X
> (code-char 255)
  #\%rubout
> (char-upcase #\a)
  #\A
> (char-upcase #\A)
  #\A
> (char-upcase #\0)
  #\0
> (char-downcase #\Z)
  #\z
> (char-downcase #\%rubout)
  #\%rubout
```

13.2 Zeichenketten

Vektoren (eindimensionale Arrays) und Listen bilden in Common LISP den Typ SEQUENCE. Charakteristisch für Objekte dieses Typs ist, daß sie aus einer Folge von einfachen oder selbst wieder komplexen Objekten bestehen. Strings sind eindimensionale Vektoren, die Zeichen vom Typ STRING-CHAR enthalten.

Es gibt eine Reihe *generischer* SEQUENCE-Funktionen; d.h. Funktionen, die sich auf alle Objekte vom Typ SEQUENCE anwenden lassen. Zu ihnen gehören u.a. LENGTH, REVERSE, REMOVE, DELETE, SUBSEQ, CONCATENATE ..., etc. Darüberhinaus gibt es eine Reihe spezieller String-Funktionen.

Beispiel (13-3)

```
> (length "")
  0
> (length "Hallo")
  5
> (remove #\a "haha")
  "hh"
> (remove #\A "haha")
  "haha"
```

13.2.1 Zugriff auf Teile einer Zeichenkette

Um ein einzelnes Zeichen aus einer Zeichenkette zu extrahieren, verwendet man die Funktion CHAR; geht es darum, auf einen längeren Teilstring zuzugreifen, empfiehlt sich die Verwendung der *Sequence*–Funktion SUBSEQ:

CHAR *String Index* [Funktion]
Die Funktion CHAR liefert als Wert das Zeichen von *String*, auf das *Index* verweist. Wie bei allen Sequenzen in LISP trägt das erste Zeichen eines Strings den Index 0.

SUBSEQ *Sequenz Start &optional Ende* [Funktion]
Diese Funktion liefert als Wert den Abschnitt von *Sequenz*, der durch *Start* (inklusive) und *Ende* (exklusiv) begrenzt wird.

Beispiel (13-4)

```
> (char "hallo" 0)
  #\h
> (char "hallo" 4)
  #\o
> (char "hallo" 5)
  Error: ...
> (subseq "Teststring" 1)
  "eststring"
> (subseq "Teststring" 1 5)
  "ests"
```

13.2.2 Zeichenkettenprädikate

Zu den wichtigsten Zeichenkettenprädikaten gehören die Funktionen STRING=, STRING-EQUAL, STRING<, STRING>, STRING<=, STRING>= und STRING/=. Der Unterschied zwischen STRING= und STRING-EQUAL besteht darin, daß STRING-EQUAL anders als STRING= Groß- und Kleinschreibung ignoriert[2]. Die Vergleichsprädikate akzeptieren *:start1 / :end1* und *:start2 / :end2* Argumente, durch die die Teile der Zeichenketten spezifiziert werden können, die miteinander verglichen werden sollen. Scheitert eines dieser Prädikate, evaluiert es zu NIL; sonst zu dem Index des ersten Zeichens, durch das sich beide Zeichenketten unterscheiden.

Beispiel (13-5)

```
> (string= "test" "Test")
  NIL
> (string-equal "test" "Test")
  T
> (string/= "test" "Test")
  0
> (string/= "test" "Test" :start1 1 :start2 1)
  NIL
> (string< "vergleichen" "vergleich")
  NIL
> (string< "vergleich" "vergleichen")
  9
```

[2] Gleiches gilt für die Funktionen STRING-LESSP, STRING-GREATERP, ...

```
> (string<= "vergleichen" "vergleich" :end1 9)
  9
```

13.2.3 Generierung und Veränderung von Zeichenketten

Zur Generierung von Zeichenketten stehen die MAKE-STRING und die STRING Funktion zur Verfügung. Der Unterschied zwischen beiden Funktionen besteht darin, daß STRING Objekte eines anderen Typs in Zeichenketten umwandelt, während MAKE-STRING Zeichenketten beliebiger Länge erzeugt, die aus n Vorkommen eines Zeichens bestehen.

STRING	*Objekt*	[Funktion]
Die Funktion STRING erwartet als Argument einen String, ein Symbol oder ein Zeichen und liefert als Wert einen String. Wenn es sich bei dem Argument um einen String handelt, bildet die Funktion ihn auf sich ab; bei einem Symbol evaluiert sie zu seinem Print-Name, und ein Zeichen konvertiert sie in einen String der Länge 1.		

MAKE-STRING	*Länge &key :initial-element*	[Funktion]
Die Funktion liefert als Wert eine Zeichenkette der Länge *länge*, die aus dem durch das *:initial-element* Argument spezifizierten Zeichen besteht. Wird kein *:initial-element* Argument spezifiziert, ist das Ergebnis implementationsspezifisch.		

Beispiel (13-6)

```
> (make-string 0)
  ""
> (make-string 5 :initial-element #\x)
  "xxxxx"
> (make-string 5)
  "^^^^^"              ; Allegro-Common-LISP
> (string "hallo")
  "hallo"
> (string 'hallo)
  "HALLO"              ; print-name des Symbols HALLO
> (string #\h)
```

"h"

Die folgenden TRIM-Funktionen erlauben es, bestimmte Zeichen aus einem String zu entfernen:

STRING-TRIM *Zeichensammlung Zeichenkette* [Funktion]
Die Funktion STRING-TRIM liefert als Wert eine Zeichenkette, die sich durch Löschung aller Vorkommen der Zeichen aus *Zeichensammlung* am Anfang und Ende von *Zeichenkette* ergibt. Als *Zeichensammlung* kann jedes Objekt vom Typ SEQUENCE dienen, das auschließlich Zeichen enthält.

Die Funktionen STRING-LEFT-TRIM und STRING-RIGHT-TRIM entfernen diese Zeichen nur am Anfang bzw. Ende der Zeichenkette.

Beispiel (13-7)

```
> (string-trim '(#\space #\*) "    * Stern *   ")
  "Stern"
> (string-left-trim '(#\space #\*) "    * Stern *   ")
  "Stern *   "
> (string-right-trim '(#\space #\*) "    * Stern *   ")
  "   * Stern"
```

Die Funktionen STRING-UPCASE und STRING-DOWNCASE ermöglichen es, eine Zeichenkette vollständig bzw. teilweise in eine nur aus Groß- / Kleinbuchstaben bestehende Zeichenkette zu konvertieren. STRING-CAPITALIZE generiert eine Zeichenkette, in der jedes Wort mit einem Großbuchstaben beginnt (gefolgt von Kleinbuchstaben). Alle drei Funktionen akzeptieren ein *:start*- bzw. *:end*-Argument.

Beispiel (13-8)

```
> (string-upcase "ein kleiner teststring")
  "EIN KLEINER TESTSTRING"
> (string-capitalize "ein kleiner teststring")
  "Ein Kleiner Teststring"
> (string-downcase "EIN KLEINER TESTSTRING" :start 4 :end 11)
  "EIN kleiner TESTSTRING"
```

Die SEQUENCE-Funktion CONCATENATE ermöglicht es, eine beliebige Folge von Objekten eines SEQUENCE-Typs zu einem Objekt zusammenzufassen:

CONCATENATE *Ergebnis-Typ* *&rest Sequences* [Funktion]
Die Funktion erzeugt ein Objekt vom Typ *Ergebnis-Typ*, der durch Verknüpfung der als Argumente spezifizierten Objekte entsteht.

Beispiel (13-9)

```
> (concatenate 'string "Ein " "kleiner" "Test")
  "Ein kleinerTest"
> (concatenate 'list '(A) '() '(B C))
  (A B C)
> (concatenate 'string #\e #\i #\n)
  ERROR ...
```

13.3 Erzeugung von möglichen Worten über Mustern

Das folgende Programm erwartet als Eingabe eine Liste mit beliebig vielen Vorkommen der Symbole VOKAL bzw. KONSONANT und generiert alle Strings, die sich über diesem Muster bilden lassen:

```
;;; Variablendeklaration

(defvar *vokale* '(#\a #\e #\i #\o #\u))

(defvar *konsonanten*
        '(#\b #\c #\d #\f #\g #\h #\j #\k #\l #\m #\n #\p #\q #\r #\s
          #\t #\v #\w #\x #\y #\z))

;;; Programm

;;;  GENERIERE-WORTE
;;;  Diese Funktion erzeugt eine Liste, die alle Sequenzen erzeugt, die sich über dem als
;;;  Parameter übergebenen Muster bilden lassen.
;;;  Der Aufruf (generiere-worte '(vokal konsonant)) erzeugt eine Liste aller Paarlisten, die
;;;  als erstes Element einen Vokal und als zweites Element einen Konsonanten enthalten.
(defun generiere-worte (muster)
  (if (endp muster)
      ()
      (kombiniere (auswerten (first muster))
                  (generiere-worte (rest muster)))))

;;; AUSWERTEN
;;; Die Funktion setzt Muster-Symbole in Mengen von Strings der Länge 1 um:
(defun auswerten (symbol)
  (if (eq symbol 'vokal)
      (char-to-string *vokale*)
      (char-to-string *konsonanten*)))

;;; CHAR-TO-STRING
;;; Konvertierung von Zeichen in Strings der Länge 1.
(defun char-to-string (zeichenmenge)
  (mapcar #'string zeichenmenge))

;;;  KOMBINIERE
;;;  Diese Funktion kombiniert jeden String der Stringmenge STR-SET1, die Strings der
;;;  Länge 1 enthält, mit allen Strings aus STR-SET2, d.h. den bislang schon gebildeten
;;;  Strings.
(defun kombiniere (str-set1 str-set2)
  (cond ((endp str-set2) str-set1)
        ((endp str-set1) nil)
        (t (append (verbinde (first str-set1) str-set2)
                   (kombiniere (rest str-set1) str-set2)))))
```

```
;;; VERBINDE
;;; Die Funktion kombiniert einen String mit allen bislang gebildeten Strings.
(defun verbinde (str str-liste)
  (mapcar #'(lambda (x)
              (concatenate 'string str x))
          str-liste))
```

Kapitel 14

Vektoren, Arrays & Hash-Tabellen

14.1 Vektoren und Arrays

In Common-LISP kann ein Array prinzipiell eine beliebige Zahl (inkl. 0) von Dimensionen besitzen. Eindimensionale Arrays werden *Vektoren* genannt. Vektoren, die nur Objekte vom Typ STRING-CHAR enthalten dürfen, werden *Zeichenketten* ("strings") genannt.

Es wird zwischen *allgemeinen Arrays* ("general arrays") und *speziellen Arrays* ("special arrays") unterschieden:

- In allgemeinen Arrays kann jedes beliebige LISP-Objekt gespeichert werden;
- in speziellen Arrays nur Objekte eines bestimmten Typs.

Der Vorteil von Arrays gegenüber Listen liegt darin, daß die Zugriffszeit auf alle im Array gespeicherten Objekte annährend konstant ist. Die größere Effizienz ist allerdings mit einer deutlich eingeschränkten Flexibilität verbunden: So ist es im allgemeinen nicht möglich, die Größe eines einmal definierten Arrays nachträglich zu ändern.

14.1.1 Generierung von Arrays

Arrays werden durch die Funktion MAKE-ARRAY generiert:

MAKE-ARRAY *Dimensionen* *Optionen* [Funktion]
Diese Funktion ermöglicht es, beliebige Arrays zu generieren. *Dimensionen* ist eine Liste nicht-negativer Zahlen: Jede dieser Zahl spezifiziert die Größe einer Dimension des Arrays. Wenn *Dimensionen* leer ist, wird ein null-dimensionaler Array erzeugt.

Durch weitere Angaben (in Form von *keyword*-Argumenten) können bestimmte Eigenschaften des erzeugten Arrays festgelegt werden:

- *:element-type*
  Als Argument kann T (Default-Wert) oder ein Typbezeichner spezifiziert werden; T erzeugt einen allgemeinen Array; ein Typbezeichner einen speziellen Array, der nur Objekte des spezifizierten Typs enthalten darf.

- *:initial-element*
  Als Argument kann jedes LISP-Objekt spezifiziert werden, das innerhalb des zu erzeugenden Arrays gespeichert werden darf. Es wird benutzt, um alle Felder des Arrays zu initialisieren. Wird kein *:initial-element* Argument spezifiziert, ist der Anfangsinhalt der Felder undefiniert (Ausnahme: *:initial-contents* Argument)

- *:initial-contents*
  Dieses Argument erlaubt es, die Felder des Arrays individuell zu initialisieren. Es sollte eine Liste sein, deren Länge mit der Zahl der Dimensionen des Arrays übereinstimmt.

- *:adjustable*
  Durch dieses Argument kann festgelegt werden, daß sich die Größe des erzeugten Arrays dynamisch verändern läßt (Default–Wert: Nil).[1]

Die maximale Zahl der Dimensionen, die maximale Größe einer Dimension und die maximale Feldzahl eines Arrays schwankt von Implementation zu Implementation. Entsprechende Informationen liefern die Systemkonstanten ARRAY-RANK-LIMIT, ARRAY-DIMENSION-LIMIT und ARRAY-TOTAL-SIZE-LIMIT.

Ein Vektor kann durch Verwendung der Funktion VECTOR erzeugt werden:

VECTOR	*Objekt**	[Funktion]
Diese Funktion bietet eine einfache Möglichkeit, Vektoren mit einem vordefinierten Inhalt zu erzeugen: Das Resultat ist ein Vektor, der genau die als Argumente von VEKTOR spezifizierten Objekte enthält.		

Da Vektoren eindimensionale Arrays sind, gilt:

$$\begin{array}{c}(\text{vektor } a_1\ a_2 \ldots a_n)\\ \equiv\\ (\text{make-array (list } n) \text{ :element-type t :initial-contents (list } a_1\ a_2 \ldots a_n))\end{array}$$

Beispiel (14-1)

> (setq array1 (make-array '(10)))

[1] Die Dimensionen eines justierbaren Arrays können durch Verwendung der Funktion ADJUST-ARRAY <*Array*> <*Dimensionen*> geändert werden.

```
   ;; Durch diese Form wird ein eindimensionaler allgemeiner Array mit 10
   ;; Feldern erzeugt.
   ;; Die Initialiserung der Felder erfolgt implementationsabhängig.
   #1A(NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)
> (setq array2 (make-array '(3 3) :element-type #'numberp :initial-element 10))
   ;; ARRAY2 ist ein zweidimensionaler Array, dessen 3 X 3 Felder mit "10"
   ;; initialisiert werden.
   #2A((10 10 10) (10 10 10) (10 10 10))
> (setq array3
        (make-array '(4 2 3) :initial-contents '(((a b c) (1 2 3)) ((d e f) (3 1 2))
                                                 ((g h i) (2 3 1)) ((j k l) (0 0 0)))))
   #3A(((A B C) (1 2 3)) ((D E F) (3 1 2)) ((G H I) (2 3 1)) ((J K L) (0 0 0)))
> (setq v1 (vector 'jim 'knopf 'und 'die 'wilde 13))
   #1A(JIM KNOPF UND DIE WILDE 13)
```

14.1.2 Zugriff auf Array-Komponenten

Die Funktion AREF kann sowohl dazu verwendet werden, um in einem Array gespeicherte Objekt zu extrahieren, wie in Kombination mit SETF neue Objekte im Array abzulegen:

AREF	*Array* *Indices**	[Funktion]
Diese Funktion liefert als Wert das Objekt in dem durch *Indices* spezifizierten Feld von *Array*. Die Zahl der Indices muß mit der Zahl der Dimensionen von *Array* übereinstimmen und ihr Wert innerhalb der Größe der Dimensionen liegen.		

Beispiel (14-2)

```
> (aref array1 8)
  NIL
> (aref array1 10)
  ;; Da der Index des ersten Feldes 0 ist, gibt es in ARRAY1 kein Feld mit
  ;; dem Index 10.
  ERROR: array-index out of range
> (setf (aref array1 0) 99)
  99
> array1
  #1A(99 NIL NIL NIL NIL NIL NIL NIL NIL NIL)
```

```
;;      0  1    2    3    4    5    6    7    8    9
> (setf (aref array2 1 1) 'versuchsfahrzeug)
;; Da bei der Generierung von ARRAY2 festgelegt wurde, das der Array
;; nur Zahlen enthalten soll, führt dieser Aufruf zu einer Fehlermeldung.
ERROR ...
> (setf (aref array2 1 1) (* 3 9))
27
> array2
#2A((10 10 10) (10 27 10) (10 10 10))
;;       00 01 02   10 11  12   20 21 22
```

14.1.3 Array-Informationen

Die folgenden fünf Funktionen ermöglichen es festzustellen, was für Objekte in einem Array gespeichert werden können, wieviele Felder bzw. Dimensionen ein Array besitzt und wie groß die einzelnen Dimensionen sind:

ARRAY-ELEMENT-TYPE	*Array*	[Funktion]
Die Funktion liefert als Wert einen Typbezeichner der spezifiziert, welche Objekte in *Array* gespeichert werden dürfen.		

ARRAY-RANK	*Array*	[Funktion]
Wert dieser Funktion ist eine nicht-negative Zahl, die die Zahl der Dimensionen von *Array* bezeichnet.		

ARRAY-DIMENSION	*Array* *Dimensions-Nr*	[Funktion]
Die Funktion liefert eine nicht-negative Zahl als Wert, die die Größe der durch *Dimensions-Nr* bestimmten Dimension von *Array* bezeichnet.		

ARRAY-DIMENSIONS	*Array*	[Funktion]
Die Funktion liefert eine Liste von Zahlen, die die Größe der Dimensionen von *Array* beschreibt.		

ARRAY-TOTAL-SIZE	*Array*	[Funktion]
Wert dieser Funktion ist die Zahl aller Felder von *Array*.		

Beispiel (14-3)

```
> (array-rank array3)
  3
> (array-dimension array3 1)
  2
> (array-dimensions array3)
  (4 2 3)
> (array-total-size array3)
  24
```

14.2 Hash-Tabellen

In LISP gibt es eine Reihe von Funktionen, die den Umgang mit Hash-Tabellen äußerst angenehm gestalten. Zur Generierung von Hash-Tabellen wird die MAKE-HASH-TABLE Funktion verwendet:

MAKE-HASH-TABLE *&key :test :size* [Funktion]
Das *:test*-Argument legt fest, welche Funktion zum Vergleich der Schlüssel verwendet wird. Zulässige Argumente bilden die Funktionen #'EQ, #'EQL (Default-Wert) und #'EQUAL). Durch das *:size*-Argument kann die Anfangsgröße der Hash-Tabelle festgelegt werden.

HASH-TABLE-P *Objekt* [Funktion]
Diese Funktion evaluiert zu T, gdw. *Objekt* eine Hash-Tabelle ist; sonst zu NIL.

Die Funktionen GETHASH und REMHASH ermöglichen die Generierung und Tilgung von Einträgen:

GETHASH *Schlüssel Hash-Tabelle [Default-Wert]* [Funktion]
Die Funktion liefert als Wert das unter *Schlüssel* in *Hash-Tabelle* abgelegte Objekt. Gibt es unter *Schlüssel* keinen Eintrag, evaluiert GETHASH zu *Default-Wert* bzw. zu NIL.

REMHASH *Schlüssel Hash-Tabelle* [Funktion]
REMHASH entfernt den Eintrag mit dem Schlüssel *Schlüssel* aus der Hash-Tabelle *Hashtabelle*. Die Funktion evaluiert zu T gdw. es einen Eintrag mit diesem Schlüssel gibt; sonst zu NIL.

CLRHASH *Hash-Tabelle* [Funktion]
Die Funktion entfernt alle Einträge aus *Hash-Tabelle* und evaluiert zu der geleerten Hash-Tabelle.

HASH-TABLE-COUNT *Hash-Tabelle* [Funktion]
Diese Funktion liefert die Zahl der Einträge in *Hash-Tabelle*.

Beispiel (14-4)

```
> (defvar *teilnehmerliste*)
  *TEILNEHMERLISTE*
> (setq *teilnehmerliste* (make-hash-table :size 40 :test #'equal))
  ;; Die auf diese Weise erzeugte Hashtabelle besteht aus 40 Feldern. Als
  ;; Vergleichsfunktion wird EQUAL verwendet.
  #<EQUAL hash-table with 0 entries @ #x3a8c81>
> (setf (gethash 'smith *teilnehmerliste*) '((nr 1) (vn john) (alter 39)))
  ((NR 1) (VN JOHN) (ALTER 39))
> (hash-table-p *teilnehmerliste*)
  T
> (gethash 'meyer *teilnehmerliste*)
  NIL
> (gethash 'smith *teilnehmerliste*)
  ((NR 1) (VN JOHN) (ALTER 39))
> (hash-table-count *teilnehmerliste*)
  1
> (remhash 'smith *teilnehmerliste*)
  T
> (remhash 'smith *teilnehmerliste*)
  NIL
> *teilnehmerliste*
  #<EQUAL hash-table with 0 entries @ #x3a8c81>
```

14.3 Eine kleine Shell

Das folgende Programm realisiert einen kleinen Kommando-Interpreter, der es erlaubt, den Inhalt von Textdateien auf den Bildschirm auszugeben und Dateien zu löschen, ohne das LISP-System zu verlassen.

```
;;; Variablendeklaration
;;; Die in der Shell verfügbaren Befehle werden unter der Variablen *KOMMANDOS*
;;; abgelegt. Das erste Symbol in jeder Paarliste bezeichnet ein Shell-Kommando, das
;;; zweite Symbol die Funktion, die bei Aufruf des Kommandos ausgeführt wird.
(defvar *kommandos*
        '((print-file . my-print-file)
```

```
        (delete-file . my-delete-file)
        (exit . my-exit)
        (help . my-help)
        (? . my-help)))
;;; In der Hashtabelle *KOMMANDO-TABELLE* soll unter jedem Befehl die mit ihm
;;; assozierte Funktion gespeichert werden:
(defvar *kommando-tabelle* (make-hash-table :size (length *kommandos*)))

;;; Programm
;;; INITIALIZE
;;; Diese Funktion initialziert die Kommando-Tabelle und aktiviert den Kommandointer-
;;; preter.
(defun initialize ()
  (dolist (x *kommandos*)
    ;; Speichern der mit den Befehlen assoziierten Funktionen:
    (setf (gethash (car x) *kommando-tabelle*)
          (symbol-function (cdr x))))
    ;; Ausgabe des Shell-Prompts:
    (format t "Befehl> ")
    ;; Starten des Kommandointerpreters:
    (main-loop *kommando-tabelle*))

;;; MAIN-LOOP
;;; Der Kommandointerpreter liest die nächste Eingabe: Ist es ein Shell-Kommando, wird
;;; aus der *KOMMANDO-TABELLE* der Name der auszuführenden Funktion extrahiert
;;; und diese Funktion ausgeführt. Sonst wird die Hilfsfunktion aufgerufen.
(defun main-loop (table)
  (do ((current-command (gethash (read) table) (gethash (read) table)))
      ((eq current-command #'my-exit) nil)
    (if current-command
        (funcall current-command)
        (funcall #'my-help))
    (format t "Befehl> ")))

;;; MY-PRINT-FILE
;;; Diese Funktion gibt den Inhalt einer Textdatei zeilenweise auf dem Bildschirm aus.
;;; Die Funktion PROBE-FILE prüft, ob die angegebene Datei existiert.
;;; GENSYM generiert ein neues, bislang nicht benutztes Symbol.
(defun my-print-file ()
  (format t "~%Wie heißt die Datei, die Sie sehen wollen?
             (Type return after the name.)~%")
```

```
        (let ((name (read-line)))
          (if (probe-file name)
              (with-open-file (file-stream name :direction :input)
                ;; als EOF-Marke wird ein sonst nicht verwendetes Symbol generiert:
                (let ((eof (gensym)))
                  (do ((current-line (read-line file-stream nil eof)
                                     (read-line file-stream nil eof)))
                      ((eq current-line eof) nil)
                    (format t "~A~%" current-line))))
              (format t "Es gibt keine Datei mit dem Namen~a~%" name))))

;;;  MY-DELETE-FILE
;;;  Die Funktion erlaubt es, Dateien zu löschen.
(defun my-delete-file ()
  (format t "~%Name der Datei, die geloescht werden soll?
             (Beenden Sie die Eingabe <RETURN>.)~%")
  (let ((name (read-line)))
    (if (probe-file name)
        ;; Die Funktion Y-OR-N-P wartet auf eine Y/N-Antwort
        (when (y-or-n-p (format t "~%Löschen der Datei~A? " name))
          ;; DELETE löscht die spezifizierte Datei:
          (delete-file name))
        (format t "Es gibt keine Datei mit dem Namen~a~%" name))))

;;; MY-EXIT
;;; Verlassen des Kommandointerpreters.
(defun my-exit ()
  "bye")

;;; MY-HELP
;;; Die Funktion zeigt die verfügbaren Kommandos an.
(defun my-help ()
  (format t "~%Folgende Kommandos können verwendet werden:~%")
  (dolist (x *kommandos*)
    (format t "~(~a~)~%" (car x))))
```

Beispiel (14-5)

```
> (initialize)
Befehl> help
Folgende Kommandos können verwendet werden:
```

```
PRINT–FILE
DELETE–FILE
EXIT
HELP
?
Befehl> delete-file
Name der Datei, die geloescht werden soll?
(Beenden Sie die Eingabe mit <RETURN>)
text1.old
Loeschen der Datei TEXT1.OLD? y
T
Befehl> exit
NIL
```

Kapitel 15

Strukturen

Ähnlich wie in den meisten prozeduralen Sprachen ist es auch Common LISP möglich, strukturierte Objekte zu generieren. Diese Objekte werden als *Strukturen* bezeichnet und bestehen wie z.B. ein PASCAL-*Record* aus einer Folge elementarer oder komplexer Komponenten.

Die Definition einer Struktur kann als Definition eines eigenen Datentyps aufgefaßt werden: Durch die Definition einer Struktur werden folgende Funktionen automatisch generiert:

1. Selektorfunktionen, die den Zugriff auf die einzelnen Komponenten von Objekten dieses Typs gestatten;
2. je eine Funktion zur Generierung und zum Kopieren von Objekten dieses Typs und
3. ein Typ-Prädikat, das nur auf Objekte dieses Typs zutrifft.

15.1 Motivation

Angenommen, man steht vor dem Problem, ein LISP-Programm zu entwickeln, das das Verhalten von Pkws im Straßenverkehr simulieren soll. Für die Beschreibung der Fahrzeuge sollen folgende Eigenschaften relevant sein:

(i). *x-Position*

(ii). *y-Position*

(iii). *geschwindigkeit*

(iv). *antriebsart* und

(v). *gewicht*.

Vor die Aufgabe gestellt, eine geeignete Repräsentation für die Pkws finden zu müssen, könnte man auf die Idee kommen, ein Fahrzeug durch eine Liste mit fünf Elementen zu repräsentieren:

$$\text{PKW}_\text{i}=(\textit{x-Position}_\text{i}\ \textit{y-Position}_\text{i}\ \textit{geschwindigkeit}_\text{i}\ \textit{antriebsart}_\text{i}\ \textit{gewicht}_\text{i})$$

d.h. die X-Position eines Fahrzeugs wird durch das CAR der es repräsentierenden Liste dargestellt, die Y-Position durch das CADR der Liste, usw.

Die Nachteile dieser Repräsentationsform sind offensichtlich: die Abbildung einer Eigenschaft auf das i-te Element einer Liste (z.B. *(third PKW) = Geschwindigkeit*) ist willkürlich und kann leicht vergessen werden. Änderungen der Repräsentation der Objekte kann dazu zwingen, das Progamm überall dort zu ändern, wo auf eine oder mehrere Eigenschaften von Objekten Bezug genommen wird.

Es gibt verschiedene Möglichkeiten, diese Mängel zu beheben: Wenn man die Listenrepräsentation der PKWs nicht aufgeben will, dann sollte man, wie wir wissen, geeignete Selektoren, Modifikatoren und Konstruktoren definieren.

Beispiel (15-1)

```
;;; Selektoren
(defun x-position (pkw)
  (first pkw))
(defun y-position (pkw)
  (second pkw))
...

;;; Modifikator
(defun aendere-x-position (pkw neuer-wert)
  (setf (first pkw) neuer-wert)
...

;;; Generator
(defun generiere-pkw (&key (x-position 0) (y-position 0) (geschwindigkeit 0)
                           (antriebsart 0) (gewicht 0))
  (list x-position y-position geschwindigkeit antriebsart gewicht))
```

Die Verwendung dieser bzw. ähnlicher Funktionen erhöht die Transparenz und die Wartungsfreundlichkeit des Programms beträchtlich: Änderungen in der Repräsentation der Objekte haben nur noch lokale Konsequenzen in dem Sinne, daß nicht beliebige Teile des Programms, sondern ausschließlich diese Funktionen, die als einzige direkt auf die zur Repräsentation der PKWs gewählte Datenstruktur zugreifen können, zu ändern sind.

Eine andere Möglichkeit bietet die Verwendung von *Strukturen*:

```
(defstruct
  pkw              ; Name der Struktur
  x-position       ; 1. Komponente
  y-position       ; 2. Komponente
  geschwindigkeit  :
  antriebsart      :
  gewicht)         ; 5. Komponente
```

Die Evaluierung dieser DEFSTRUCT-Form hat folgende Konsequenzen: Es wird ein Datentyp PKW definiert und festgelegt, daß jedes Objekt dieses Typs aus fünf Komponenten besteht. Außerdem werden folgende Funktionen generiert:

1. die Selektorfunktionen PKW-X-POSITION, PKW-Y-POSITION, PKW-GESCHWINDIGKEIT, PKW-ANTRIEBSART und PKW-GEWICHT;
2. ein einstelliges Prädikat PKW-P: es liefert T, gdw. sein Argument ein Pkw ist (sonst NIL)[1];
3. eine Funktion MAKE-PKW, die jedesmal, wenn sie aufgerufen wird, ein Objekt vom Typ PKW erzeugt, und
4. eine einstellige Funktion COPY-PKW, die als Resultat eine Kopie des Objekts liefert, das ihr Argument bildet.

Nach der Definition einer Struktur können die mit ihr assoziierten Selektorfunktionen in Kombination mit SETF zur Veränderung der Komponenten von Objekten diesen Typs verwendet werden.

Beispiel (15-2)

```
;;; Soll ein (Struktur-)Objekt erzeugt werden, können die bei der Definition der Struktur
;;; verwendeten Komponentennamen als key-Argumente verwendet werden, um gewünsch-
;;; te Werte für die einzelnen Komponenten zu spezifizieren:
> (setq auto1 (make-pkw :x-position 47 :y-position 82 :geschwindigkeit 0
                        :antriebsart 'diesel :gewicht 1400))
  #s(PKW :X-POSITION 47 :Y-POSITION 82 :GESCHWINDIGKEIT 0
         :ANTRIEBSART 'DIESEL :GEWICHT 1400)
```

[1] Außerdem wird das Symbol PKW zu einem legalen Datentypnamen: Es ist jetzt ein zulässiges Argument für die TYPEP-Funktion.

```
> (pkw-antriebsart auto1)
  DIESEL
> (setf (pkw-geschwindigkeit auto1) 123)
  123
> auto1
  #s(PKW :X-POSITION 47 :Y-POSITION 82 :GESCHWINDIGKEIT 123
         :ANTRIEBSART 'DIESEL :GEWICHT 1400)
> (setq auto2 (copy-pkw auto1))
  #s(PKW :X-POSITION 47 :Y-POSITION 82 :GESCHWINDIGKEIT 123
         :ANTRIEBSART 'DIESEL :GEWICHT 1400)
> (equal auto1 auto2)
  NIL
> (pkw-p auto1)
  T
```

15.2 Der DEFSTRUCT-Macro

Strukturen werden, wie wir schon sahen, durch das DEFSTRUCT-Makro definiert. Die Syntax und Semantik dieses Makros sind (wie nicht anders zu erwarten war) gewöhnungsbedürftig:

DEFSTRUCT	{*Name* | (*Name* {*Option*}$^+$)} [*Dokumentations-String*] {*Slot-Beschreibung*}*
[Makro]	
Durch die Evaluierung einer DEFSTRUCT-Form wird eine Struktur *Name* definiert, und die zuvor beschriebenen Funktionen werden generiert.	

Ein wenig übersichtlicher notiert, läßt sich die Syntax einer DEFSTRUCT-Form wie folgt darstellen:

```
(defstruct (name option-1 option-2 ... option-m)
           dokumentationsstring
           komponente-1-beschreibung
           komponente-2-beschreibung
           ...
           komponente-n-beschreibung)
```

Bei *name* muß es sich um ein Symbol handeln; dieses Symbol wird zum Bezeichner für den neuen Datentyp. Eine DEFSTRUCT-Form evaluiert zu *name*. Wenn keine Option spezifiziert wird, kann statt (*name*) einfach *name* verwendet werden.

Jede *Komponenten-Beschreibung* besteht aus einer Liste, die den Komponentennamen und optional einen Anfangswert und Komponentenparameter enthält:

(*komponenten-name anfangswert*
komponentenoption-name-1 *komponentenoption-wert-1*
komponentenoption-name-2 *komponentenoption-wert-2*
...
komponentenoption-name-n *komponentenoption-wert-n*)

Als Komponentennamen dürfen nur Symbole verwendet werden. Wenn für einen Komponente kein Anfangswert vereinbart wird, dann kann die Komponentenbeschreibung einfach aus dem Komponentennamen bestehen. Allerdings ist in diesem Fall der Wert dieser Komponente undefiniert (d.h. es wird ein implementationsabhängiger Anfangswert verwendet).

15.2.1 Komponentenoptionen

Komponenten-Optionen werden wie auch die DEFSTRUCT-Optionen durch ein Schlüsselwort/Wert-Paar spezifiziert. Es gibt zwei Komponenten-Optionen:

- *:type*
  Durch diese Option kann der Datentyp der Objekte deklariert werden, die in dieser Komponente gespeichert werden können. Als Argument dieses Schlüsselwortes muß ein Typbezeichner verwendet werden.

- *:read-only*
  Evaluiert das auf dieses Schlüsselwort folgende Argument zu einem anderen Wert als NIL, kann der Inhalt der Komponente nicht mehr verändert werden; anderenfalls hat es keine Wirkung. Diese Option kann nur verwendet werden, wenn ein Anfangswert spezifiziert wurde.

Beispiel (15-3)

```
> (defstruct lkw
    (x-position 0 :type integer)
    (y-position 0 :type integer)
    (geschwindigkeit 0 :type (integer 0 120))
    (antriebsart 'diesel :read-only T)
    gewicht)
  LKW
> (setq auto3 (make-lkw :gewicht 7500))
  #s(LKW :X-POSITION 0 :Y-POSITION 0 :GESCHWINDIGKEIT 0
```

```
                    :ANTRIEBSART 'DIESEL :GEWICHT 7500)
> (setf (lkw-antriebsart auto3) 'benzin)
  ERROR ...
> (setf (lkw-geschwindigkeit auto3) 240)
  ERROR ...
```

15.2.2 DEFSTRUCT-Optionen

Es gibt eine ganze Reihe DEFSTRUCT-Optionen; die wichtigsten fünf sind:

- *:conc-name*
  Normalerweise werden die Namen der Selektorfunktionen durch Verbindung von Strukturnamen und Komponentennamen gebildet; z.B. PKW-GEWICHT (Strukturname-Komponentenname). Die *:conc-name* Option erlaubt es, einen anderen Präfix zu spezifizieren, der dann anstelle des Strukturnamens zur Bildung der Selektornamen verwendet wird.

  ```
  (defstruct (pkw (:conc-name 'auto)) ...)
  ```

  deklariert eine Struktur PKW mit den Selektorfunktionen AUTO-X-POSITION, AUTO-Y-POSITION, etc. Wenn das Argument von *:conc-name* zu NIL evaluiert, wird kein Präfix zur Namensbildung verwendet; d.h. Selektor- und Komponentennamen sind in diesem Fall identisch.

- *:constructor*
  Mit diesem Schlüsselwort kann ein anderer Präfix zur Bildung des Namens der mit einer Struktur assoziierten Konstrukturfunktion gewählt werden (normalerweise: MAKE-*Strukturname*). Evaluiert das Argument dieses Schlüsselwortes zu NIL, wird keine Konstruktorfunktion generiert.

- *:copier*
  Diese Option erlaubt die Vereinbarung eines anderen Namens für die Kopierfunktion als *COPY-Strukturname*. Evaluiert das Argument dieses Schlüsselwortes zu NIL, wird keine Kopierfunktion generiert.

- *:predicate*
  Diese Option erlaubt die Vereinbarung eines anderen Namens für das mit der Struktur assoziierte Typenprädikat; normalerweise: *Strukturname-p*. Evaluiert das Argument von :predicate zu NIL, wird kein Typ-Prädikat erzeugt.

- *:include*
  Diese Option ermöglicht es, eine neue Struktur als Extension einer bereits definierten Struktur zu bilden (Vererbung von Komponenten).

Beispiel (15-4)

```
> (defstruct person name alter geschlecht)      ; PERSON-Struktur
  PERSON
;;; Eine weitere Struktur zur Repräsentation von Astronauten, die sich von der PERSON-
;;; Struktur durch zwei zusätzliche Komponenten helm-groesse und lieblingsstern unter-
;;; scheidet, kann wie folgt definiert werden:
> (defstruct (astronaut (:include person) (:conc-name astro-))
    helm-groesse (lieblingsstern 'sirius))
  ASTRONAUT
;;; Die Verwendung der :include-Option sorgt dafür, daß die Struktur ASTRONAUT die
;;; drei Komponenten der Struktur PERSON 'erbt'.
;;; Das hat zur Folge, daß die entsprechenden Selektorfunktionen für PERSON auch für
;;; Objekte des Typs ASTRONAUT verwendet werden können. Außerdem werden noch
;;; die Selektoren ASTRO-NAME, ASTRO-ALTER, ATRO-GESCHLECHT generiert, die
;;; nur auf Objekten des Typs ASTRONAUT, nicht aber auf Objekten des Typs PERSON
;;; operieren können. Ebenso trifft PERSON-P auf alle Objekte vom Typ PERSON und
;;; ASTRONAUT zu; ASTRONAUT-P aber nur auf die Objekte vom Typ ASTRONAUT.
> (setq person1 (make-person :name 'hugo :alter 99 :geschlecht 'männlich))
  #s(PERSON :NAME HUGO :ALTER 99 :GESCHLECHT MÄNNLICH)
> (setf person2
        (make-astronaut :name 'helga :alter 33 :geschlecht 'weiblich :helm-groesse 27))
  #s(ASTRONAUT :NAME HELGA :ALTER 33 :GESCHLECHT WEIBLICH
               :HELM-GROESSE 27 :LIEBLINGSSTERN SIRIUS)
> (person-name person1)
  HUGO
> (person-name person2)
  HELGA
> (astro-name person1)
  ERROR ...
> (astro-name person2)
  HELGA
> (person-p person2)
  T
> (astronaut-p person2)
  T
> (astronaut-p person1)
```

NIL

Wie dieses Beispiel zeigt, ermöglicht die *:include*-Option die Vererbung von Komponenten. Zu beachten ist allerdings, daß in der Definition einer Struktur die *:include*-Option höchstens einmal spezifiziert werden darf, d.h. Mehrfachvererbung (*multiple inheritance*) ist nicht möglich.

Allerdings können durch die Angabe von Komponentenbeschreibungen innerhalb der *:include*-Option die Anfangswerte der Struktur, von der Komponenten übernommen werden, verändert werden; d.h. in diesem Fall ist folgende Syntax zu verwenden:

(:include *name*
komponente-1-beschreibung
komponente-2-beschreibung
...
komponente-n-beschreibung)

Es gelten folgende Restriktionen: geerbte *read-only* Komponenten bleiben *read-only* Komponenten; enthält eine geerbte Komponente eine Typrestriktion, kann sie übernommen werden oder ein Subtyp dieses Typs vereinbart werden. So kann z.B. die Typrestriktion (*:type number*) durch die Angabe (*:type integer*) weiter eingeschränkt werden.

15.3 Merkmalsstrukturen und Kategorien

In unifikationsbasierten Grammatikformalismen werden syntaktische und lexikalische Kategorien durch Merkmalsstrukturen repräsentiert. Das folgende Programm illustriert, wie solche Kategorien mit Hilfe von Strukturen repräsentiert werden können.

15.3.1 Vorüberlegungen

Wir gehen von folgenden Annahmen aus: Eine Merkmalsstruktur besteht aus einer Menge von Attribut-Wert Paaren. Es werden zwei Typen von Kategorien unterschieden: *Haupt*- und *Neben*-Kategorien; weiterhin gibt es vier Klassen von Hauptkategorien: nominale, verbale, adjektivische und präpositionale Kategorien.

Kategorien
  Hauptkategorien
    nominal
    verbal
    adjektivisch
    präpositional
  Nebenkategorien
    Artikel

Interjektionen
. . . .

Wir verwenden folgende Attribute:

Name	*Wertebereich*	*Kommentar*
typ	{major, minor}	unterscheidet Haupt- und Nebenkategorien
minor_typ	{art, inj . . . }	unterscheidet verschiedene Nebenkategorien
n	{+, -}	definiert Hauptkategorientypen
v	{+, -}	definiert Hauptkategorientypen
bar	{0, 1, 2}	lexikalische versus syntaktische Kategorien
vform	{fin, inf, psp, prp}	verschiedene Verbformen
nform	{norm, null}	unterscheidet Nomen und Expletivia
pform	{auf, an, . . . }	
subj	{+, -}	unterscheidet V und VP von S
aux	{+, -}	Hilfsverb versus Vollverb
per	{1, 2, 3}	
num	{sg, pl}	
temp	{prae, past}	
kas	{1, 2, 3, 4}	
gen	{m, f, n}	
def	{+, -}	bestimmter versus unbestimmter Artikel

15.3.2 Implementierung

```
;;; (1) Globale Variable(n)

;;; Default-Wert für nicht spezifizierte Merkmale:
(defvar *undefiniert* '?)

;;; (2) Kategorien

;;; Grundkategorie:
(defstruct kategorie (typ *undefiniert* :type symbol))

;;; Hauptkategorie
;;; Übernimmt die TYP-Komponente von KATEGORIE mit MAJOR als festem Wert.
;;; Neue Komponenten: N, V, BAR.
(defstruct (major-kategorie (:include kategorie (typ 'major :read-only t)))
  (n *undefiniert*) (v *undefiniert*) (bar 0 :type (integer 0 2)))

;;; Nebenkategorie
;;; MINOR wird als Wert für TYP fixiert.
;;; Neue Komponente: MINOR-TYP.
```

```
(defstruct (minor-kategorie (:include kategorie (typ 'minor :read-only t)))
  (minor_typ *undefiniert*))

;;; Die vier Hauptkategorien:

;;; (a) Verbale Kategorien
;;; {<V, +>, <N, ->}
;;; Neue Komponenten: VFORM, SUBJ(-), AUX(-), PER, NUM, TEMP.
(defstruct (verb-kategorie
             (:include major-kategorie (n '- :read-only t) (v '+ :read-only t)))
  (vform *undefiniert*) (subj '-) (aux '-) (per *undefiniert*) (num *undefiniert*)
  (temp *undefiniert*))

;;; (b) Nominale Kategorien
;;; {<V, ->, <N, +>}
;;; Neue Komponenten: NFORM, KAS, NUM, GEN.
(defstruct (nom-kategorie
             (:include major-kategorie (n '+ :read-only t) (v '- :read-only t)))
  (nform *undefiniert*) (kas *undefiniert*) (num *undefiniert*) (gen *undefiniert*))

;;; (c) präpositionale Kategorien
;;; {<V, ->, <N, ->}
;;; Neue Komponenten: PFORM, KAS.
(defstruct (praep-kategorie
             (:include major-kategorie (n '- :read-only t) (v '- :read-only t)))
  (pform *undefiniert*) (kas *undefiniert*))

;;; (d) Adjektivische Kategorien
;;; {<V, +>, <N, ->}
;;; Neue Komponenten: KAS, NUM, GEN.
(defstruct (adj-kategorie
             (:include major-kategorie (n '+ :read-only t) (v '+ :read-only t))
             (:print-function print-adjektiv))
  (kas *undefiniert*) (num *undefiniert*) (gen *undefiniert*))

;;; Zwei Nebenkategorien:

;;; Artikel-Kategorie
;;; MINOR_TYP = ART
;;; Neue Komponenten: DEF, KAS, NUM, GEN.
(defstruct (artikel (:include minor-kategorie (minor_typ 'art :read-only t)))
  (def *undefiniert*) (kas *undefiniert*) (num *undefiniert*) (gen *undefiniert*))

;;; Interjektionen-Kategorie
```

```
;;; MINOR_TYP = INJ
(defstruct (interjektionen (:include minor-kategorie (minor_typ 'inj :read-only t))))
```

Bei der Definiton der ADJ-KATEGORIE haben wir neben den bekannten Optionen die *:print-function* Option verwendet, die es erlaubt, eine Funktion anzugeben, die für eine gewünschte Darstellung der Objekte dieses Typs sorgt. Die folgende Funktion sorgt für eine ansprechende Darstellung von adjektivischen Kategorien:

```
(defun print-adjektiv (struktur stream depth)
  (format stream "~%< n: +~% v: +~% kas: ~A~% num: ~A~% gen: ~A >~%"
    (adj-kategorie-kas struktur)
    (adj-kategorie-num struktur)
    (adj-kategorie-gen struktur)))
```

Beispiel (15-5)

```
> (setq kuchen (make-nom-kategorie :nform 'norm :gen 'mas))
  #s(KUCHEN :TYP MAJOR :N + :V - :BAR 0 :NFORM NORM :KAS ? :NUM ?
            :GEN MAS)
> (setq alte (make-adj-kategorie :kas 1 :num 'sg :gen 'f))

  < n: +
    v: +
    kas: 1
    num: SG
    gen: F >
```

Kapitel 16

Weitere Hilfsmittel

In diesem abschließenden Kapitel werden wir zwei Komponenten von LISP-Systemen beschreiben, die für die Entwicklung größerer Programme unentbehrlich sind: der Compiler und die Funktionen die zusammengenommen häufig als *Debugging–Tools* bezeichnet werden. Leider sind diese beiden Systemkomponenten nicht standartisiert, d.h. die implementationspezifischen Differenzen zwischen verschiedenen LISP-Systemen können in diesem Bereich gravierend sein.

16.1 Compilieren von Funktionen und Programmen

Der Compiler eines LISP-System übersetzt Lispfunktionen und -programme in einen Kode, der eine schnellere Ausführung der Funktionen bzw. Programme erlaubt (Faktor 3–15) gilt als üblich). Normalerweise wird ein Lispprogramm nicht in ein Maschinensprachprogramm, sondern in einen sogenannten Zwischenkode übersetzt, was zur Folge hat, das auch die kompilierten Programme nur innerhalb des LISP-Systems ausgeführt werden können.

Prinzipiell sollte das kompilierte Programm diesselbe Semantik aufweisen, wie das ihm zugrundeliegende (interpretierte) LISP-Programm. Es gibt allerdings Fälle, wie wir sehen werden, in denen das Verhalten des kompilierten Programms sich von dem des interpretierten Programms deutlich unterscheidet. Die implementationsunabängige Schnittstelle zum Compiler bilden die beiden Funktionen COMPILE und COMPILE-FILE:

COMPILE	*Name* *&optional Definition*	[Funktion]
Die Funktion compiliert eine einzelne Funktion. Als Argument kann entweder ein Symbol oder ein Symbol und ein Lambda–Form angegeben werden. Im ersten Fall wird die als Funktionsbindung von *Name* etablierte Funktion kompiliert; im zweiten Fall wird der Lambda–Form kompiliert und das Resultat in der Funktionszelle von *Name* gespeichert.		

COMPILE-FILE	*Pfadname* *&key :output-file*	[Funktion]
Diese Funktion kompiliert alle der in der durch *pfadname* spezifizierten Datei definierten Funktionen. Das *key*-Argument erlaubt es, den Namen der Datei festzulegen, die den Kode der kompilierten Funktionen enthält.		

Wenn die COMPILE-FILE Funktion verwendet wird, erhält die Datei, die den kompilierten Kode enthält normalerweise den Namen: *name*.fasl oder *name*.fsl, mit *fasl*/*fsl* = "**fas**t-**l**oad"-Format). Diese Dateien können genauso geladen werden wie andere Programmdateien. Es gibt Systeme, die nach kompilieren einer Datei den kompilierten Kode direkt laden.

Beispiel (16-1)
Angenommen, die Datei test.lsp enthält die folgenden beiden Funktionsdefinitionen:

```
(defun ab-&-aufbau (liste)
  (if (endp liste)
      nil
      (cons (first liste) (ab-&-aufbau (rest liste)))))
(defun fakultaet (zahl)
  (if (<= zahl 0)
      1
      (* zahl (fakultaet (- zahl 1)))))
```

```
> (load ¨test.lsp¨)
  T
> (symbol-function 'ab-&-aufbau)
  (LAMBDA (LISTE) (BLOCK AB-&-AUFBAU (IF (ENDP LISTE) NIL
    (CONS (ANFANG LISTE) (AB-&-AUFBAU (REST LISTE))))))
  ; --- Compiling file //node_27635/win/naumann/lisp/test.lsp ---
  ; Compiling FAKULTAET
  ; Compiling AB-&-AUFBAU
  ; Writing fasl file "//node_27635/win/naumann/lisp/test.fasl"
  ; Fasl write complete
> (load ¨test.fasl¨)
  T
> (symbol-function 'ab-&-aufbau)
```

```
  #<Function AB-&-AUFBAU @ #x3a8169>
> (ab-&-aufbau '(a b c))
  Error: Attempt to call #<function object ANFANG (a macro) @ #x3a8359>
  which is defined as a macro.
```

Grundsätzlich sollte man beim kompilieren von Funktionen und Programmen folgende Punkte beachten:

1. Funktionen sollten erst kompiliert werden, nachdem sie ausreichend getestet wurden. Das Verhalten von kompilierten Funktionen läßt sich durch Funktionen wie TRACE, BACKTRACE und STEP nur sehr eingeschränkt sichtbar machen.

2. Makros können genau wie Funktionen kompiliert werden. Allerdings sollte man dafür sorgen, daß sie vor den Funktionen, in denen sie genutzt werden sollen, kompiliert werden: Da beim Kompilieren der durch die Makros bezeichnete Kode in die Funktionen, die sie aufrufen, kopiert wird, kommt es sonst zu Fehlermeldungen.

   Beispiel (16-2)
   Wenn z.B. die Datei test.lsp aus dem letzten Beispiel noch folgende Makrodefinition enthält:

   ```
   (defmacro anfang (liste) '(first ,liste))
   ```

   und der Aufruf in AB-&-AUFBAU (FIRST LISTE) ersetzt wird durch (ANFANG LISTE), dann funktioniert nach dem Kompilieren der Datei alles wie gehabt, wenn die Makrdefinition am Anfang der Datei steht. Anderenfalls kommt es zu einer Fehlermeldung wie:

   Undefined Function ANFANG ...

16.1.1 Deklarationen

DECLARE	*Deklarationsform* *	[Special Form]
Durch DECLARE werden Deklarationen realisiert. Mögliche Position für Deklarationen: DEFUN, DEFMACRO, LET, LET*, DO, DO*, DOLIST, DOTIMES, ...		

- (special Var_1 Var_2 ... Var_n
- (type *typ-bezeichner* Var_1 Var_2 ... Var_n)
- (optimize ($Qualität_1$ $Wert_1$) ... ($Qualität_1$ $Wert_1$)),
  mit $Qualität_i$ *in* {speed, safety, space, compiler-speed} und $Wert_i \in \{0, 1, \ldots 3\}$.

16.2 Fehlersuche

TRACE	*Funktionsname* *	[Makro]

Die durch die angegebenen Funktionsnamen bezeichneten Funktionen werden '*getraced*'; d.h. innerhalb eines Programmaufrufs werden bei jedem Aufruf einer dieser Funktionen alle Argumente und der von der Funktion zurückgelieferte Wert in den Stream *TRACE-OUTPUT* ausgegeben. Ein Aufruf von TRACE ohne Argumente evaluiert zu der Liste der Funktionen, die zur Zeit *überwacht* werden.

UNTRACE	*Funktionsname* *	[Makro]

Alle durch die als Argumente übergebenen Funktionsnamen bezeichneten Funktionen werden nicht weiter '*getraced*'. Der Aufruf von UNTRACE ohne weitere Argumente führt dazu, daß keine Funktion weiter überwacht wird.

Beispiel (16-3)

```
> (trace fakultaet)
  (FAKULTAET)
> (trace)
  (FAKULTAET)
> (trace ab-&-aufbau)
  (AB-&-AUFBAU)
> (trace)
  (AB-&-AUFBAU FAKULTAET)
> (ab-&-aufbau '(a (b (c))))
  0:(AB-&-AUFBAU (A B C))
    1:(AB-&-AUFBAU (B C))
      2:(AB-&-AUFBAU (C))
        3: (AB-&-AUFBAU NIL)
        3: returned NIL
      2: returned (C)
    1: returned (B C)
  0: returned (A B C)
  (A (B (C)))
> (fakultaet 4)
  0:(FAKULTAET 4)
```

```
    1:(FAKULTAET 3)
      2:(FAKULTAET 2)
        3:(FAKULTAET 1)
          4: (FAKULTAET 0)
          4: returned 1
        3: returned 1
      2: returned 2
    1: returned 6
  0: returned 24
  24
> (untrace)
  (AB-&-AUFBAU FAKULTAET)
> (untrace)
  NIL
```

STEP	*Form*	[Makro]
Eine STEP-Form evaluiert und liefert den Wert von *Form*. Allerdings wird *Form* im "singlestep"-Modus evaluiert, und es werden verschiedene (systemabhängige) Optionen zur Verfügung gestellt, die Evaluierung zu beeinflussen.		

Beispiel (16-4)

```
> (step (ab-&-aufbau '(a b c)))[1]
#10: (AB-&-AUFBAU (QUOTE (A B C)))
Step> help
?           - Display help text for the Stepper commands.
BACKTRACE   - Displays a backtrace of the current form.
DEBUG       - Invoke the debugger.
EVALUATE    - Evaluate a specified form.
FINISH      - Complete evaluation without the stepper.
HELP        - Display help text for the Stepper commands.
OVER        - Evaluate the current form with stepping disabled.
QUIT        - Exits the stepper.
RETURN      - Return the specified values.
SHOW        - Display the current form without abbreviation.
STEP        - Single step the current form.
UP          - Return to the containing form.
```

```
Step> step
: #17: (BLOCK AB-&-AUFBAU (IF (ENDP LISTE) NIL (CONS (FIRST LISTE)
       (AB-&-AUFBAU (REST LISTE)))))
Step> step
: : #22: (IF (ENDP LISTE) NIL (CONS (FIRST LISTE) (AB-&-AUFBAU (REST LISTE))))
Step> step
: : : #27: (ENDP LISTE)
Step> [RETURN]
: : : : #32: LISTE =¿ (A B C)
: : : #27 => NIL
: : : #27: (CONS (FIRST LISTE) (AB-&-AUFBAU (REST LISTE)))
Step> backtrace
(CONS (FIRST LISTE) (AB-&-AUFBAU (REST LISTE)))
(IF (ENDP LISTE) NIL (CONS (FIRST LISTE) (AB-&-AUFBAU (REST LISTE))))
(BLOCK AB-&-AUFBAU (IF (ENDP LISTE) NIL (CONS (FIRST LISTE)
  (AB-&-AUFBAU (REST LISTE)))))
(AB-&-AUFBAU (QUOTE (A B C)))
Step> step
: : : : #32: (FIRST LISTE)
Step> step
: : : : : #37: LISTE => (A B C)
: : : : #32 => A
: : : : #32: (AB-&-AUFBAU (REST LISTE))
Step> [RETURN]
: : : : : #32 => (B C)
: : : #27 => (A B C)
: : #22 => (A B C)
: #17 => (A B C)
#10 => (A B C)
(A B C)
```

[1] In diesem Beispiel wird der Stepper des VAX-LISP zugrundegelegt.

16.3 Diverses

TIME	*Form*	[Makro]
Auch TIME-Formen evaluieren *form* und liefern als Wert den Wert dieser Form. Per Seiteneffekt werden allerdings noch in den Stream *TRACE-OUTPUT* Informationen über die zur Evaluation benötigte Zeit (tatsächliche Zeit, CPU-Zeit, ...) ausgegeben.		

Beispiel (16-5)

```
> (time)
  Autoload: TIME (FUNCTION cell) from D:\GCL3_1\LISPLIB\EVALTIME.F2S
  Elapsed Time: 0:00.00
  NIL
> (time t)
  Elapsed Time: 0:00.00
  T
> (defun fakultaet (n) (if (<= n 0) 1 (* n (- n 1))))
  FAKULTAET
> (time (fakultaet 3))
  Elapsed Time: 0:00.00
  6
> (time (fakultaet 1000))
  Elapsed Time: 0:00.00
  999000
> (defun ab-&-aufbau (list)
    (cond ((endp list) nil)
          ((symbolp (first list))
           (cons (first list) (ab-&-aufbau (rest list))))
          (t (cons (ab-&-aufbau (first list)) (ab-&-aufbau (rest list))))))
  AB-&-AUFBAU
> (time (ab-&-aufbau (quote ((a) ((b (c nil ((nil))) (d)))))))
  Elapsed Time: 0:00.00
  ((A) ((B (C NIL ((NIL))) (D))))
> (time (ab-&-aufbau '((a) ((b (c ((((((((((((((((((nil)))))))))))))))))))((nil))) (d))))))
  Elapsed Time: 0:00.05
```

((A) ((B (C ((((((((((((((((NIL)))))))))))))))) ((NIL))) (D))))

DESCRIBE	*Objekt*	[Funktion]
DESCRIBE gibt in den Stream *STANDARD-OUTPUT* Informationen über *objekt* aus. DESCRIBE liefert keinen Wert zurück.		

Beispiel (16-6)

```
> (describe (quote a))
  A is an internal symbol in package USER.
  Its value is unbound.
  Its function definition is unbound.
  Its property list is empty.
> (SETQ A 123)
  123
> (DESCRIBE A)
  123 is an odd fixnum.
> (describe (quote a))
  A is an internal symbol in package USER.
  Its value is: 123.
  Its function definition is unbound.
  Its property list is empty.
> (defun a (x) (> x x))
  A
> (describe (quote a))
  A is an internal symbol in package USER.
  Its value is: 123.
  Its function definition is
  #<LISP::SCANNED (LAMBDA (X) (DECLARE) (BLOCK A (* X X)))>
  Its property list contains:
```

Property: :LAMBDA-LISTS, Value: ((:FUNCTION X))

DRIBBLE	*&optional Pfadname*	[Funktion]
Eine (DRIBBLE *pfadname*)-Form hat zur Folge, daß alle mit *STANDARD-INPUT* & *STANDARD-OUTPUT* verbundenen Ausgaben auch in die durch *pfadname* bezeichnete Datei geschrieben werden. Existiert eine Datei mit diesem Namen, wird sie überschrieben; sonst angelegt. Mit (DRIBBLE) wird diese Datei dann wieder geschlossen.		

16.4 Reguläre Ausdrücke

Das folgende Beispiel stammt aus: Charniak, Riesbeck, McDermott & Meehan (1987), S.10-23.

```
(defun match-p (string dfa-function)
  (funcall dfa-function string))

(setq dfa1
      #'(lambda (string)
          (let ((n 0) (end (length string)))
            (labels ((state-0 (c)
                       (cond ((null c) nil)
                             ((char= c #\a)
                              (state-1 (next)))
                             ((char= c #\b)
                              (state-0 (next)))
                             (t nil)))
                     (state-1 (c)
                       (cond ((null c) nil)
                             ((char= c #\a)
                              (state-1 (next)))
                             ((char= c #\b)
                              (state-2 (next)))
                             (t nil)))
                     (state-2 (c)
                       (cond ((null c) nil)
                             ((char= c #\a)
                              (state-1 (next)))
```

```
                ((char= c #\b)
                 (state-3 (next)))
                (t nil)))
        (state-3 (c)
          (cond ((null c) t)
                ((char= c #\a)
                 (state-1 (next)))
                ((char= c #\b)
                 (state-0 (next)))
                (t nil)))
        (next ()
          (if (= n end)
              nil
              (prog1
                (char string n)
                (setq n (+ n 1))))))
                (state-0 (next))))))
```

```
> (match-p "aabb" dfa1)
  T
> (match-p "aabba" dfa1)
  NIL
```

```
(defmacro iterate (name specs . body)
  `(labels ((,name ,(mapcar #'first specs) ,@body))
     (,name ,@(mapcar #'second specs))))

(defun match-p-v2 (string dfa)
  (let ((end (length string)))
    (iterate match ((state-index 0) (string-index 0))
      (let ((state (nth state-index dfa)))
        (if (= string-index end)
            (first state)
            (let ((transition (assoc (char string string-index)
                                     (rest state)
                                     :test #'char=)))
              (if (null transition)
                  nil
                  (match (rest transition) (+ string-index 1)))))))))

(setq dfa2
```

```
'((nil (#\a . 1) (#\b . 0))
  (nil (#\a . 1) (#\b . 2))
  (nil (#\a . 1) (#\b . 3))
  (t (#\a . 1) (#\b . 0))))
```

```
> (match-p-v2 "aabb" dfa2)
T
> (match-p-v2 "aabba" dfa2)
NIL
```

Anhang A

Die wichtigsten EMACS-Kommandos

A.1 Aufrufen und Verlassen des Editors

emacs	*dateiname* (optional)
ECS x	*exit* oder
Ctrl-x Ctrl-c	Emacs verlassen

A.2 Bewegen im Text

ESC-f	forward-word
ESC-b	backward-word
Ctrl-a	beginning-of-line
Ctrl-e	end-of-line
POS 1	beginning-of-window
ENDE	end-of-window
Ctrl-v	next-page
ESC v	previous-page
ESC <	beginning-of-file
ESC >	end-of-file

A.3 Loeschen und Einfuegen

del	delete-last-character
Ctrl-d	delete-next-character

ESC-d	kill-next-word
Ctrl-k	kill-to-end-of-line
Alt-k	kill-to-beginning-of-line
Ctrl-m	newline
Ctrl-@	set-mark
Ctrl-w	kill-region
ESC w	copy-region
Ctrl-y	yank

A.4 Suchen und Ersetzen

Ctrl-s	search-forward
Ctrl-r	search-reverse
ESC q	query-replace-string

A.5 File-Handling

C-x C-v	visit-file
Ctrl-x Ctrl-w	write-file
Ctrl-x s	save-file
Ctrl-x Ctrl-b	list-buffers
Ctrl-x b	select-buffer

A.6 Weitere Kommandos

Ctrl-l	redraw-display
Ctrl-g	Ausführung eines Kommandos abbrechen
ESC x	erweiterte Kommandos

Anhang B

Funktionale Programmiersprachen

B.1 Funktionale und imperative Programmiersprachen

B.1.1 Variablen

In imperativen Programmiersprachen (IPGS) besteht ein Programm aus einer Folge von Anweisungen:

$$
\begin{array}{l}
<Anweisung_1> \\
<Anweisung_2> \\
\quad \vdots \quad \vdots \quad \vdots \\
<Anweisung_n>
\end{array}
$$

Einen wichtigen Anweisungstyp bilden *Zuweisungen*, durch die der Wert einer Variablen geändert wird:

$$<Name> := <Ausdruck>$$

Jede Anweisung kann von Variablen Gebrauch machen, deren Wert durch vorangegangene Anweisungen festgelegt wurde. Da der Wert einer Variablen prinzipiell beliebig oft verändert werden darf, gilt:

In imperativen Programmiersprachen kann derselbe Name mit unterschiedlichen Werten assoziiert werden.

Funktionale Programmiersprachen (FPGS) dagegen basieren auf dem Konzept des Funktionsaufrufs. Ein Programm besteht aus einem (beliebig tief verschachtelten) Funktionsaufruf:

$$<Funktion_1>(<Funktion_2>(<Funktion_3> \ldots) \ldots)$$

Variablen werden nur als formale Parameter der Funktionen verwendet, die beim Funktionsaufruf an die Werte der aktuellen Argumente gebunden werden. Da es keine Anweisungen

gibt, durch die der Wert einer Variablen verändert wird, gilt:

In funktionalen Programmiersprachen kann eine Variable nur mit einem Wert assoziiert werden.

B.1.2 Ausführungsreihenfolge

In IPGS ist die Reihenfolge, in der die Anweisungen, aus denen das Programm besteht, ausgeführt werden, ganz entscheidend. Besonders deutlich wird diese Tatsache, wenn Anweisungen verwendet werden, die Seiteneffekte erzielen: In diesem Fall kann die Änderung der Ausführungsreihenfolge die Semantik des Programms vollständig verändern:

Beispiel (2-1)
Der Wert zweier Variablen V_1, V_2 soll vertauscht werden:

$$\begin{array}{ll} W & := V_1 \\ V_1 & := V_2 \\ V_2 & := W \end{array}$$

Verändert man die Reihenfolge der Anweisungen, erhält man völlig andere Resultate:

$$\begin{array}{ll} V_1 & := V_2 \\ W & := V_1 \\ V_2 & := W \end{array} \qquad\qquad \begin{array}{ll} W & := V_1 \\ V_2 & := W \\ V_1 & := V_2 \end{array}$$

In imperativen Programmiersprachen determiniert die Ausführungsreihenfolge die Semantik der Programme.

In FPGs ist die Ausführungsreihenfolge, da es keine Wertzuweisungen gibt und eine Variable immer mit demselben Wert assoziiert bleibt, für die Semantik der Programme irrelevant; obwohl natürlich die Ausdrücke, aus denen ein Programm besteht, in einer bestimmten Reihenfolge ausgewertet werden.

In funktionalen Programmiersprachen beeinflußt die Ausführungsreihenfolge nicht die Semantik der Programme.

B.1.3 Iteration und Rekursion

Wenn eine Anweisungssequenz mehrfach auszuführen ist, ist es in einer IPGS nicht erforderlich, die Anweisungssequenz zu dublizieren: es gibt geeignete Kontrollstrukturen, die eine mehrfache Ausführung dieser Sequenz ermöglichen. Infolgedessen werden die in der Sequenz verwendeten Variablen bei der Programmausführung mit verschiedenen Werten assoziiert:

Beispiel (2-2)
Die Addition der Element eines Arrays A mit n Feldern:

```
I := 0;
SUMME := 0;
WHILE I < N DO
    BEGIN
        I := I + 1;
        SUMME := SUMME + A[I]
    END
```

In imperativen Programmiersprachen können dieselben Variablen durch wiederholte Ausführung von Anweisungen mit neuen Werten assoziiert werden.

Da in FPGS eine Variable nicht an verschiedene Werte gebunden werden kann, entfallen iterative Lösungen und es liegt nahe, stattdessen rekursive Funktionen zu verwenden, die bei jedem Aufruf neue Parameter generieren, die an die neuen Werte gebunden werden:

Beispiel (2-3)

```
FUNCTION SUMME(A:ARRAY [1..N] OF INTEGER; I,N:INTEGER):INTEGER;
    BEGIN
        IF I > N THEN
            SUMME := 0
        ELSE
            SUMME := A[I] + SUMME(A, I+1, N)
    END
```

In funktionalen Programmiersprachen werden neue Werte mit neuen Werten durch rekursive Funktionsaufrufe assoziiert.

B.1.4 Funktionen als Werte

In vielen IPGS (z.B: PASCAL) ist es möglich Funktione zu definieren, die funktionale Argumente akzeptieren; funktionale Werte dagegen sind nicht zulässig.
FPGS sind in dieser Hinsicht 'liberaler':

> In funktionalen Programmiersprachen ist es möglich, Funktionen zu definieren, die selbst wieder Funktionen als Argumente nehmen und als Werte liefern.

B.2 Die Entwicklung funktionaler Programmiersprachen

Die theoretischen Grundlagen für die Entwicklung funktionaler Programmiersprachen bildeten Untersuchungen zur Theorie der Rerechenbarkeit (bes. Kleenes Theorie der rekursiven Funktionen und Churchs λ-Kalkül). Die zunehmende Anwendung von Computern zu Beginn der 60iger Jahre führte zu einem verstärkten Interesse an den praktischen Implikationen dieser theoretischen Arbeiten.

Als eine der ersten funktionalen Sprachen wurde Anfang der 60iger Jahre von J.McCarthy LISP entwickelt, eine Programmiersprache, die funktionale und imperative Elemente miteinander verbindet. Mitte der 60iger Jahre schlugen Landin & Strachey vor, das λ-Kalkül als Grundlage für die Entwicklung imperativer Programmiersprachen zu verwenden.
Landin selbst entwickelte auf der Basis einer operationalen Interpretation des λ-Kalküls einen abstrakten Interpreter (die SECD-Maschine), den er dann verwendete, um einen abstrakten Interpreter für ALGOL 60 zu formulieren.

Zu den wichtigsten neben LISP entwickelten funktionalen und partiell funktionalen Sprachen gehören:

POP-2 > POP11 -> POPLOG (LISP + PROLOG)
SASL > KRL
> MIRANDA
Hope
ML

Literaturverzeichnis

[CRMM87] E. Charniak, C.K. Riesbeck, D.V. McDermott, and J.R. Meehan. *Artificial Intelligence Programming*. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 2. edition, 1987.

[DR] D.Vorberg and R.Göbel. Rekursionsschemata als problemlösepläne. Technical report, Fachbereich Psychologie, Universität Marburg, ???

[GM89] G. Gazdar and C. Mellish. *Natural Language Processing in LISP*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

[Kee89] Sonya E. Keene. *Object-oriented Programming in Comon LISP: A Programmers Guide to CLOS*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

[Mic88] Greg Michaelson. *An Introduction to Functional Programming through Lambda Calculus*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1988.

[Sol85] E. Soloway. From problems to programs via plans: The content and structure of knowledge for introductory lisp programming. *Journal of Educational Computing Research*, 1(2):157–172, 1985.

[Ste90] Guy L. Steele. *LISP. The Language*. Digital Press, Bedford, Massachusetts, 2. edition, 1990.

[Win88] Patrick H. Winston. *LISP*. Addison-Wesley Publishing Company, Reading, Massachusetts, 3. edition, 1988.

Index