Konstrukte
∩
Konstruktionen
∩
Konstruktionsempfehlungen

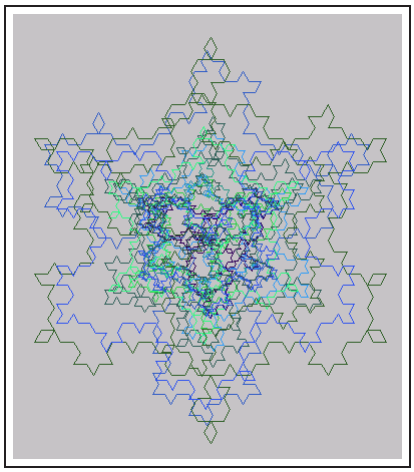
Programmieren in *LISP*

(Scheme^{R6})

Hinrich E. G. Bonin[1]

[1]Prof. Dr. rer. publ. Dipl.-Ing. Dipl.-Wirtsch.-Ing. Hinrich E. G. **Bonin** lehrte bis Ende März 2010 „Informatik in der Öffentlichen Verwaltung“ an der Leuphana Universität Lüneburg, Institut für Wirtschaftsinformatik (IWI), Email: Hinrich@hegb.de, Adresse: An der Eulenburg 6, D-21391 Reppenstedt, Germany.

Zusammenfassung

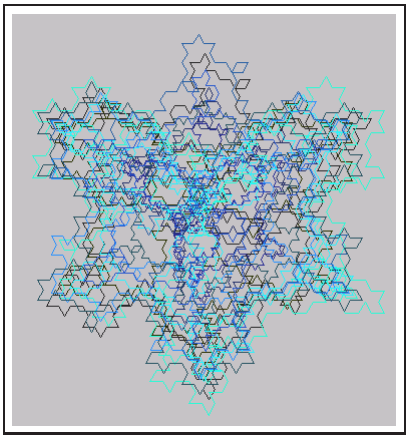


Dieses Buch zeigt programmieren als einen systematischen, disziplinierten Konstruktionsvorgang. Dazu wird LISP (Akronym für „*LIst Processing*“) aus klassischen und neuen Blickwinkeln erläutert. LISP ist eine bewährte Sprache, geschaffen für die Bewältigung komplexer Aufgaben und prädestiniert für den Umgang mit Symbolen, abgeleitet aus dem λ-Kalkül und geprägt durch ihre Dualität von *Daten* und *Programm*. LISP ermöglicht mit seinem modernen Dialekt *Scheme* ohne Ballast ein imperativ-, funktions- und objekt-geprägtes programmieren.

Dieses Buch zeigt zweckmäßige Bausteine (Kapitel I), skizziert bewährte Konstruktionen (Kapitel II) und gibt praxisnahe Konstruktionsempfehlungen (Kapitell III). Es basiert auf meiner langjährigen Lehrtätigkeit an verschiedenen Hochschulen[1] und ist daher dominant geprägt von Ideen und Bedürfnissen der Studierenden.

[1] Hochschule Bremerhaven, Universität Bremen, Universität Hannover, Fachhochschule Nordostniedersachen und Leuphana Universität Lüneburg.

Vorwort



Nützliche Produkte entstehen kaum per Zufall. Sie erfordern einen sorgsam geplanten Herstellungsprozess, wenn es gilt, sie wirtschaftlich, in gewünschter Qualität termingerecht zu produzieren. Ein zielführendes Vorgehen bedingt fundierte Kenntnisse über mögliche Bausteine (Konstrukte), bewährte Konstruktionen und nützliche Handlungsweisen (Konstruktionsempfehlungen). Dieses Buch vermittelt solche Kenntnisse. Es wendet sich an alle, die Schritt für Schritt, anhand von leicht nachvollziebaren Beispielen in *LISt Processing* (LISP), fundiertes Wissen über Programme gewinnen wollen. Aufbauend auf der klassischen LISP-Basis werden moderne Konzepte, wie die Objekt-Orientierung und die Aufteilung auf mehrere Akteure (z. B. *Multithreading*), ausführlich behandelt und somit verstehbar.

Dieses Buch zum weiten Feld des *programmierens in LISP* ist primär ein Produkt meiner Lehrtätigkeit an verschiedenen Hochschulen und daher von den Ideen, Hinweisen und Korrekturen vieler Jahrgänge von Studierenden bestimmt. Als LISP-Dialekt dient jetzt *Scheme* in der Form des *Revised*[6] *Report on the Algorithmic Language Scheme* (↪ [170]). Die Ausführungen sind zum Teil eine Aktualisierung und Fortschreibung meines Buches „*Software-Konstruktion mit LISP*“ (↪ [14]). Aufgrund der Veranstaltung „*Advanced Programming*“ im Leuphana-Bachelor (Minor Angewandte Informatik), im Wintersemester 2009/10 sind didaktisch bewährte Beispiele modernisiert und an die Weiterentwicklungen im LISP-Bereich angepasst worden; z. B. dient nun das moderne *PLT-Scheme* [2] mit der Entwicklungsumgebung von *DrScheme* (Version 4.2.3; Dez. 2009) sowie Bibliotheken von *mzscheme* als „LISP-Leitdialekt“.

Sicherlich lässt sich LISP nüchtern und sachlich erläutern, ohne vom Glanz der „LISP-Philosophe“ zu schwärmen. Dieses Buch versucht je-

[2] *PLT-Scheme* ↪`http://www.plt-scheme.org/` (Zugriff: 24-Oct-2009)

doch, die Begeisterung des Autors für das moderne LISP zu verdeutlichen. Erst wenn die „*Lots of Insidious Silly Parentheses*“[3] in ästhetische Konstrukte umschlagen, kommt zur Zweckmäßigkeit auch noch die Freude.

Dieses Buch ist konzipiert als ein *Arbeitsbuch zum Selbststudium und für Lehrveranstaltungen*. Der LISP-Neuling möge es Abschnitt für Abschnitt sequentiell durcharbeiten. Der LISP-Vorbelastete kann zunächst die Zusammenfassungen der einzelnen Abschnitte studieren. Anhand ihrer *zusätzlichen* charakteristischen Beispiele ist individuell entscheidbar, ob der Inhalt des jeweiligen Abschnittes schon bekannt ist. Damit das Buch auch später als Nachschlagewerk hilfreich ist, enthält es sowohl Vorwärts- wie Rückwärts-Verweise.

Ebensowenig wie z. B. Autofahren allein aus Büchern erlernbar ist, wird niemand zum „*LISP wizard*“ (deutsch: Hexenmeister) durch Nachlesen von erläuterten Beispielen. Das intensive Durchdenken der Beispiele im Dialog mit einem LISP–System vermittelt jedoch, um im Bild zu bleiben, unstrittig die Kenntnis der Straßenververkehrsordnung und ermöglicht ein erfolgreiches Teilnehmen am Verkehrsgeschehen. Für diesen Lernprozess wünsche ich Ihnen viel Freude.

[3] Eine spöttische Interpretation von LISP: viele heimtückische und blödsinnige Klammern.

Danksagung: Für das große Interesse und die Durchsicht von vorhergehenden Fassungen danke ich den vielen Mitwirkenden: Studierenden und Kollegen. Stellvertretend für alle sei namentlich Kollege Prof. Dr. *Fevzi Belli* erwähnt. Für die Bearbeitung dieser Fassung gebührt Frau Dipl.-math. *Karin Dening-Bratfisch*, Herrn Dipl.-Kfm. *Norbert Tschritter* und Herrn Dipl.-Ing. *Christian Wagner* mein besonderer Dank.

Lüneburg,
den (1989…1991)…Oktober 2009 … 12. März 2010
Hinrich E. G. Bonin
’((V) e r f a s s e r)

Vereinbarungen und Notation

Allgemeines

Aus Lesbarkeitsgründen sind nur die männlichen Formulierungen genannt; die Leserinnen seien implizit berücksichtigt. So steht z. B. das Wort „Programmierer“ hier für Programmiererin und Programmierer.

Soweit wie möglich stehen deutsche Fachbegriffe. Ausnahmen bilden Begriffe, bei denen die Übersetzung nicht „eins-zu-eins“ möglich ist und/oder die englische Fassung sich durchgesetzt hat, z. B. *Flavor System* (deutsch: Aroma, Wohlgeschmack) als Bezeichnung für eine objekt-orientierte Entwicklungsumgebung. Um das Verstehen des laufenden Textes zu sichern, sind Textpassagen im Sinne von gedanklichen Einschüben abgegrenzt. Dabei treten folgende Einschübe auf:

Hinweis: < *text* >
< *text* > verweist auf eine Besonderheit oder Empfehlung.

Exkurs: < *text* >
Zum besseren Verstehen der Hintergründe erläutert < *text* > einen zusätzlichen Aspekt.

Notation für die Schnittstelle zum LISP-System

`eval>` < *sexpr* > `==>` < *sexpr* > `;Kommentar`

`eval>` Diese Zeichenfolge gibt an, dass die danachfolgenden Zeichen dem Zyklus „Lesen, Auswerten, Ergebnisausgabe“ (*READ-EVAL-PRINT*-Zyklus) übergeben werden. `eval>` steht anstelle der Aufforderung (Prompt) eines laufenden LISP-Systems, das eine Eingabe erwartet.
LISP-Systeme verwenden unterschiedliche Aufforderungen und zusätzlich sind diese oft individuell modifizierbar. Wir benutzen `eval>` (↪ [29, 137]) für Scheme. Zur Betonung eines bestimmten LISP-Dialektes ist ein Kennbuchstabe als Präfix vorangestellt.

< *sexpr* > *symbolic expression*; ein symbolischer Ausdruck (≡ LISP-Konstrukt)

`==>` Dokumentiert das Ergebnis, d. h. die nach dem Pfeil dargestellten Zeichen werden vom *PRINT*-Teil ausgegeben.

`;` Das Semikolon kennzeichnet einen Kommentar. Alle Zeichen nach dem Semikolon in einer Zeile sind im *READ-EVAL-PRINT*-Zyklus Kommentare, d. h. werden vom LISP-System nicht ausgewertet (↪ Abschnitt 3.1.2 S. 389).

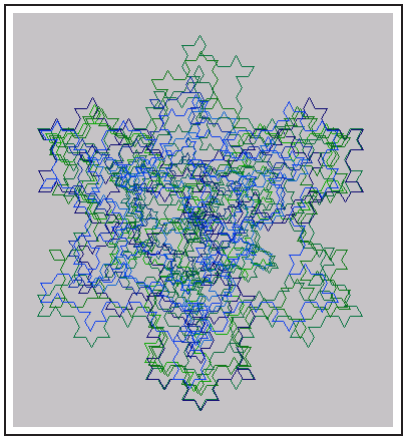
Die *Scheme*-Konstrukte könnten mit runden, eckigen und geschweiften Klammern geschrieben werden. Wir nutzen diese Differenzierung im Schriftbild nicht, sondern bleiben bei der klassischen Schreibweise mit runden Klammern.

```
eval> '(A (B C) D) ==> (A (B C) D)
eval> '{A (B C) D} ==> (A (B C) D)
eval> '[A {B C} D] ==> (A (B C) D)
```

Inhaltsverzeichnis

Kapitel 1

Konstrukte



Wie jede Sprache, so dient auch eine formal(isiert)e Sprache der Kommunikation. Aufgabe einer Programmiersprache ist die Kommunikation zwischen Menschen über Texte („Botschaften“), die ein Rechner als Arbeitsanweisungen „interpretieren“ und ausführen kann. Formulierungen in einer Programmiersprache haben einerseits Menschen, andererseits Rechner als Adressaten.

Jeder Text in einer Programmiersprache hat zumindest im Erstellungsprozess (Entwurfs- / Formulierungs- / Test-Zyklus) seinen Konstrukteur als Kommunikationspartner. Darüber hinaus wendet sich der Text an Dritte; an Mitkonstrukteure und an das Wartungs-/Pflege- und Fortentwicklungs-Personal (↪ Abbildung 1.1 S. 12). Programmieren ist damit das Erstellen von Texten aus (sprachlichen) Konstrukten, welche die „Arbeit“ eines Rechners bestimmen.

Programmiersprachen können stärker die Kommunikation mit dem Rechner oder die zwischen Programmierern unterstützen. Ein in Maschinencode notiertes Programm kann das gesamte Leistungsvermögen

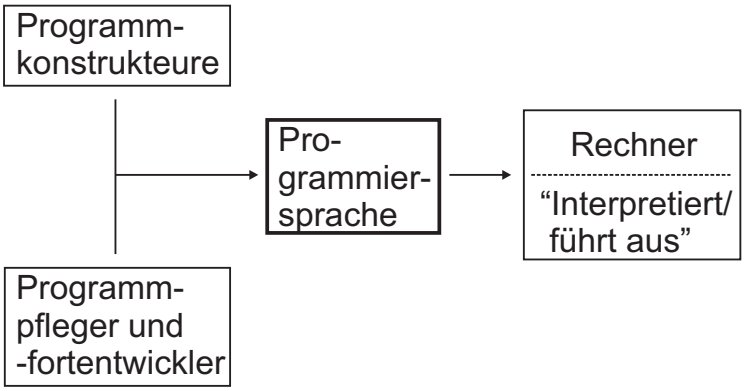
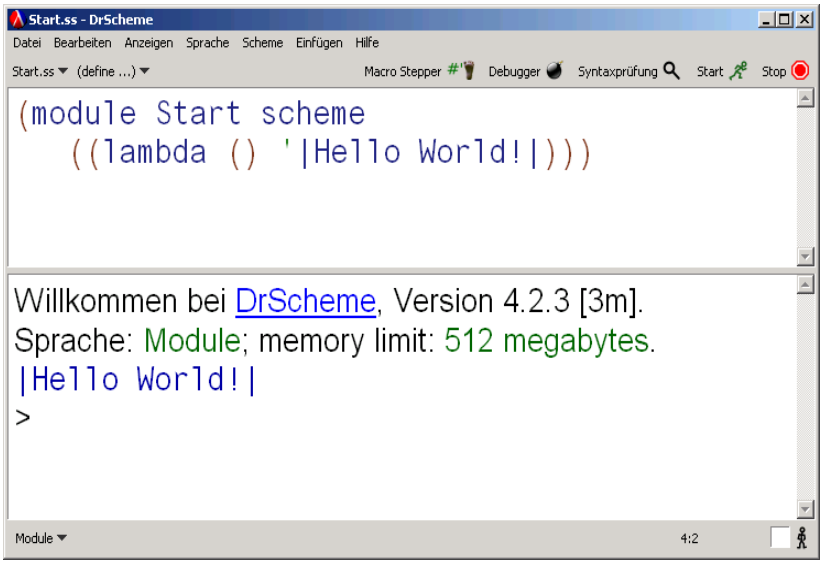


Abbildung 1.1: Kommunikationsmittel Programmiersprache



Legende:
Das `lambda`-Konstrukt, eingebettet in dem Modul `Start`, wird ausgewertet und hat als Wert ein Symbol, dessen Namen aufgrund des Leerzeichens durch ein sogenanntes Fluchtsymbole begrenzt ist. Näheres zum genutzten *PLT-Scheme*-System ↪ Abschnitt 5 S. 15.

Abbildung 1.2: Klassisches Beispiel: `Hello World!`

(„Verstehbarkeitspotential“) des Rechners effizient ausnutzen. Ein Programmierer kann ein solches Programm in der Regel nur mühsam verstehen. Ein Text, in einer Assembler-Sprache formuliert, ist im Allgemeinen aufwendiger zu durchschauen als ein äquivalenter Text, notiert in einer höheren Programmiersprache. Dafür wird der Assembler-Text als eine 1:1-Abbildung des Maschinencodes vom Rechner mit weniger Aufwand in konkrete Arbeitsschritte umgesetzt. Eine Programmiersprache ist daher stets ein Kompromiss zwischen den unterschiedlichen Anforderungen der Kommunikations-„Partner“ Mensch und Maschine.

In einer natürlichen Sprache kann ein Sachverhalt klar und leicht verständlich oder umständlich („verkompliziert“) und missverständlich beschrieben werden. Kurze Sätze mit prägnanten Begriffen, verknüpft durch eine systematische Gliederung, helfen einen Sachverhalt besser, klarer und schneller zu vermitteln als lange Sätze mit stark verschachtelten Bezügen. In jeder Programmiersprache, also auch in *LISt Processing* (LISP), steht man vor einem ähnlichen Formulierungsproblem. Es können leicht durchschaubare (transparente) oder undurchschaubare bzw. nur sehr schwer durchschaubare Texte notiert werden. Es geht daher um die Wahl geeigneter Begriffe (Sprachkonstrukte) und um ihre Kombination zu einem transparenten Text, der den Sachverhalt (die Problemlösung) so einfach wie möglich vermittelt.

Üblicherweise beginnt der Einstieg in eine formale (Programmier-)Sprache mit der Ausgabe der Meldung „`Hello World!`“ auf dem Bildschirm. Dieser „bewährten“ Informatik-Gepflogenheit folgen wir, obwohl Sie die Lösung möglicherweise nur im Ansatz verstehen. So zeigt Abbildung 1.2 S. 12 vorab dazu LISP-typisch die Applikation einer anonymen Funktion. Weil weltweit alle LISP-Programmierer zunächst einmal „`Hello World!`“ notieren, ist die hier gezeigte Kostprobe von allen anderen zu unterscheiden. Wir betten sie deshalb in einem eigenen Modul ein.

Im ersten Kapitel erläutern wir Konstrukte der Programmiersprache LISP und ihre Handhabung mit dem Ziel, ein „zweckmäßiges“ (transparentes und effizientes) Programm zu konstruieren (↪ Abschnitt 1.1 S. 14). Programmieren wird als Lösen einer Konstruktionsaufgabe aufgefasst (↪ Abschnitt 1.2 S. 50), wobei der Programmtext (Quellcodetext) die Konstruktion einerseits für den Programmierer (Transparenzaspekt) und andererseits für den Rechner (Effizienzsaspekt) dokumentiert.

Bestimmte Konstruktionen sind für eine Programmiersprache charakteristisch. Für LISP sind es die Konstruktionen der rekursiven Problemlösungsmethode. Nicht jede weitverbreitete Programmiersprache verfügt über Konstrukte für rekursive Lösungen; z. B. besitzt COBOL (*Common Business Oriented Language*) selbst im derzeit gültigen Standard[1] über kein Konstrukt zur Formulierung rekursiver Definitionen; es müssen

[1] American National Standard COBOL X3.23-1985

iterative (Ersatz-)Lösungen gewählt werden. Obwohl LISP iterative Konstrukte umfasst unterstützt LISP rekursive Lösungen in besonderem Maße. Die Rekursion wird daher ausführlich erörtert (↪ Abschnitt 1.3 S. 113).

1.1 Einführung: Handhabung von Konstrukten

Heute ist LISP eine universelle Programmiersprache, eine dialogorientierte Software-Entwicklungsumgebung, eine Menge vordefinierter Bausteine (**Konstrukte**) und ein Formalismus zur Spezifikation (Präzisierung) eines Problems.

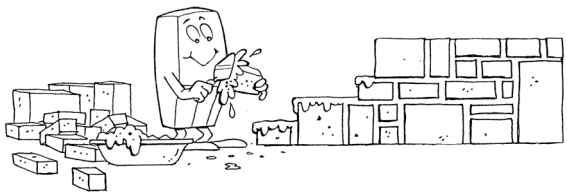
Die Ursprungsfassung konzipierte *John McCarthy* in den Jahren 1956 – 1958 am *Massachusetts Institute of Technology* (MIT), Cambridge, Massachusetts (↪ [123, 124]). Dabei konnte er auf frühere Arbeiten zurückgreifen, insbesondere auf die erste Listen-Verarbeitungs-Sprache mit dem Namen *IPL* (*Information-Processing Language*), die Mitte der fünfziger Jahre von *Herbert A. Simon*, *A. Newell* und *J. C. Shaw* entwickelt wurde. Die lange Historie hat zu einer großen Anzahl von LISP-Dialekten geführt (↪ Abschnitt A.2 S. 462) und zu mehreren Versuchen, einen Standard zu definieren und diesen durchzusetzen, z. B. *Common LISP* (↪ [174]) oder *Scheme*[2] (↪ [151]). Trotz der Vielfalt existierender Dialekte gibt es einen gemeinsamen LISP-Kern, das sogenannte „*Pure LISP*“. Typisch für diesen Kern ist die Möglichkeit, anwendungsspezifische Funktionen mit Hilfe des λ-Konstruktes (↪ Abschnitt 1.1.4 S. 35) definieren zu können.

LISP ist konzipiert für den Umgang mit Symbolen. Wie der Name *LISt Processing* besagt, ist LISP geprägt:

- durch die Liste als das Mittel zur Darstellung von Strukturen und
- durch den Prozess der Auswertung von Listen (symbolischen Ausdrücken).

Beide Kennzeichen werden erst im Sinne einer „Kostprobe“ vermittelt und in den folgenden Abschnitten unter verschiedenen Aspekten weiter vertieft. Zunächst ist es erforderlich die Syntax elementarer Konstrukte (engl.: *primitives*) zu erläutern. Danach gilt es die Semantik dieser Konstrukte zu beschreiben. Dabei geht es um die Bedeutung der elementaren Konstrukte, d. h. um die Werte, die solche Konstrukte aus (korrekten) Eingaben berechnen.

[2] Zur weiteren Standardisierung von *Scheme*-Bibliotheken gibt es seit dem *Scheme*-Workshop in Baltimore, Maryland (26-Sep-1998) im Web das Forum „*Scheme Requests for Implementation*“ (SRFI):
↪ `http://srfi.schemers.org/` (Zugriff 23-Jan-2010)



Die folgenden LISP-Beispiele sind in *PLT-Scheme* notiert und getestet.

> *PLT-Scheme is an innovative programming language that builds on a rich academic and practical tradition. It is suitable for implementation tasks ranging from scripting to application development, including GUIs, web services, etc. It includes the* **DrScheme programming environment***, a virtual machine with a just-in-time compiler, tools for creating stand-alone executables, the PLT-Scheme web server, extensive libraries, documentation for both beginners and experts, and more. It supports the creation of new programming languages through a rich, expressive syntax system. Example languages include Typed Scheme, ACL2, FrTime, Lazy Scheme, and ProfessorJ (which is a pedagogical dialect of Java) — Latest release: 4.2.2, (October 2009)*
> *(↪*`http://www.plt-scheme.org/` *(Zugriff: 24-Oct-2009)*

1.1.1 Syntax: Symbolischer Ausdruck

Die Bausteine zur Konstruktion einer Liste sind die leere Liste und (Listen-)Elemente. Ein einfaches Element kann z. B. eine Zahl oder ein Symbol sein. Solche Elemente werden auch Atome genannt. Diese Bezeichnung weist darauf hin, dass sie einerseits eine (kleinste) betrachtbare Einheit und andererseits der Stoff zum Bau komplexer Konstruktionen sind. Später zeigen wir, dass auch in LISP die Atome aus elementaren Teilchen bestehen.

Die leere Liste wird dargestellt durch eine öffnende Klammer „`(`“, unmittelbar gefolgt von einer schließenden Klammer „`)`“. Diese leere Liste wird klassischerweise durch das Symbol `nil` bzw. `NIL`[3] repräsentiert. In Scheme steht für die leere Liste das Symbol `null`. Werden Elemente in die leere Liste eingefügt, wobei die Elemente selbst wiederum Listen sein können, dann entstehen geschachtelte Listen. Das Fundament („Mutter Erde“) für solche komplexen Strukturen ist `NIL`, die leere Liste.

Die Tabelle 1.1 S. 16 zeigt Schritte zur Konstruktion einer geschachtelten Listenstruktur. Die Liste selbst und jedes Element stellen einen sogenannten symbolischen Ausdruck dar (engl.: *symbolic expression*; abgekürzt „S-Expression“ oder kürzer „*sexpr*“; Plural „*sexprs*“).

Ein symbolischer Ausdruck (*symbolic expression*) kann sein:

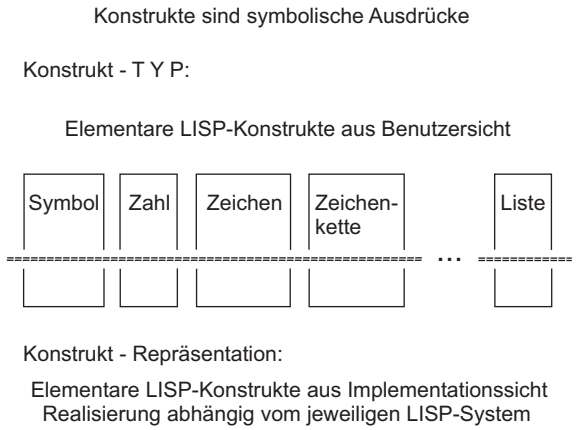
[3] Hinweis: In *PLT-Scheme* (DrScheme) ist `NIL` bzw. `nil` wie folgt zu definieren:

```
eval> (define nil (list))
eval> nil ==> ()
```

Schritte	Aktion	Erzeugte Struktur
1.	`null`, leere Liste† angelegt	`()`
2.	Element C eingefügt	`(C)`
3.	Element B vorn eingefügt	`(B C)`
4.	Element A vorn eingefügt	`(A B C)`
5.	Liste `(E F G)` an die Stelle von A gesetzt	`((E F G) B C)`
6.	Element A vorn eingefügt	`(A (E F G) B C)`

Legende:
† ≡ Klassisch wird die leere Liste mit `NIL` statt mit `null` angegeben.

Tabelle 1.1: Beispiel: Von der leeren zu einer geschachtelten Liste



Legende:
=== ≡ Betrachtungsebene; Verstehen der elementaren Konstrukte ohne Implementations-Detailkenntnisse

Abbildung 1.3: Symbolische Ausdrücke

1. ein **Symbol** (auch Literalatom genannt),
   Z. B. `A` oder `null?` oder `Meyer-AG`, notiert als ein Buchstabe oder Sonderzeichen, gefolgt von keinem oder weiteren Buchstaben oder Sonderzeichen.

2. eine **Zahl** (auch Zahlatom genannt),
   Z. B. `77` oder `+67.23`, notiert mit optionalem Vorzeichen und gefolgt von einer oder mehreren Ziffer(n) und optionalem Dezimalpunkt.

3. ein **Zeichen** (engl.: *character*),
   Z. B. `#\a`, dargestellt als Nummernzeichen # (engl.: *sharp sign*) gefolgt von einem Schrägstrich (engl.: *backslash*) und gefolgt von dem jeweiligen Zeichen.

4. eine **Zeichenkette** (engl.: *string*),
   Z. B. `"Programmiersprachen bleiben bedeutsam!"`
   notiert als doppeltes Hochkomma, gefolgt von keinem, einem oder mehreren Zeichen und abgeschlossen wieder mit einem doppelten Hochkomma.

5. eine **Liste** von symbolischen Ausdrücken und
   Z. B. `(A (B))` oder `(Abbuchen! Meyer-AG 0.10)`, notiert als öffnende Klammer gefolgt von keinem, einem oder mehreren symbolischen Ausdrücken und einer schließenden Klammer.

6. abhängig vom jeweiligen LISP-System **weitere Typen und Strukturen**.
   Z. B. Vektoren, spezielle Symbole für Boolsche Wahrheitswerte, oder Umgebungen.

Ein LISP-Programm ist ein symbolischer Ausdruck, der sich aus symbolischen Ausdrücken zusammensetzen kann, die sich wiederum aus symbolischen Ausdrücken zusammensetzen können usw. Ein LISP-Programm besteht letztlich aus zusammengesetzten symbolischen Ausdrücken einfachen Typs. Diese Typen haben Eigenschaften, die unabhängig von den Implementationsdetails des jeweiligen LISP-Systems (↪ Abbildung 1.3 S. 16) verstehbar sein sollten.

Jeder symbolische Ausdruck ist eine Datenstruktur und gleichzeitig ein syntaktisch korrektes LISP-Programm (im Sinne einer kontextfreien Analyse). Ein LISP-Programm bewirkt nichts anderes, als symbolische Ausdrücke ein- und auszugeben, neu zu konstruieren, zu modifizieren und/oder zu vernichten. Die LISP-Syntax ist durch die Regeln zur Bildung symbolischer Ausdrücke definiert. Die Liste, ein symbolischer Ausdruck, kann sich wiederum aus Listen zusammensetzen. Die LISP-Syntax ist aufgrund dieser rekursiven Definition sehr einfach (↪ Tabelle 1.9 S. 67).

Beispiele für Listen

```
(Franz Meyer-GmbH)
(Hans Otto-OHG (Abteilung 3))
(#t #f #f #t)
(() ())
((Eine) (gueltige Liste) ist dies mit Zahlatom 12)
(((((A))) (((B)))) (1 2 3 4 5 6 7 8 9))
```

Beispiel für eine unvollständige Liste

```
(Dies (ist))
  ((nicht eine)) Liste)
```

Dieser Listennotation fehlt die öffnende Klammer am Anfang!

Um zu erkennen, ob eine gültige Liste vorliegt, bedarf es der Zuordnung der öffnenden Klammern „(“ zu den schließenden Klammern „)“. Nur wenn die Zahl der öffnenden Klammern gleich der Zahl der schließenden ist, kann eine oder können mehrere Listen vorliegen. Diese Bedingung ist notwendig, jedoch nicht hinreichend, wie der folgende Fall zeigt.

```
))(Keine Liste (   ;Keine Liste,
                   ; da falsche
                   ; Klammernfolge!
```

Ein Verfahren zum Erkennen zusammengehörender Klammern ist daher erforderlich. Es stellt fest, ob es sich um eine oder mehrere Listen handelt oder keine ordnungsgemäße Liste vorliegt. Bei vielen öffnenden und schließenden Klammern ist nicht gleich überschaubar, welche Liste durch welche beiden Klammern begrenzt wird. Nicht zuletzt darauf beruht die spöttische Uminterpretation des Akronyms LISP in *Lots of Insidious Silly Parentheses*. Das folgende Beispiel verdeutlicht das Problem bei vielen Klammern.

```
 ((HIER (SIND (VIELE)(SCHOENE)))
 ((KLAMMERN)(((ZU)))((ZAEHLEN))))
```

Moderne LISP-Systeme entschärfen dieses Problem der Klammernzuordnung durch integrierte Strukturediteren, die zu einer schließenden Klammer die zugehörende öffnende Klammer hervorheben, z. B. durch Aufblinkenlassen dieser Klammer. Da beim Analysieren von LISP-Texten nicht immer eine entsprechende Rechnerunterstützung verfügbar ist, müssen die Klammern manchmal auch manuell „gezählt“ werden. Es gibt verschiedene Empfehlungen für das Vorgehen beim Zählen von Klammern (↪ [166] S. 214 ff). Die Tabelle 1.2 S. 19 skizziert ein Verfahren mit Angabe der Schachtelungstiefe von Listen in Listen.

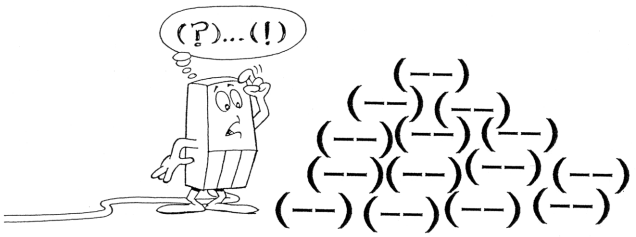
Dieses Verfahren, auf das obige Beispiel angewendet, ergibt folgende Schachtelungstiefen:

LISTENERKENNUNG	
Schritte	**Aktion**
1.	**Erste Zeichen prüfen** Ist es eine öffnende Klammer, dann setze ihre SCHACHTELUNGSTIEFE auf 1, andernfalls beende LISTENERKENNUNG, denn es ist keine Liste.
2.	**Klammern markieren** Suche zeichenweise nach Klammern, beginnend vom ersten Zeichen bis zum letzten Zeichen, und notiere die SCHACHTELUNGSTIEFE nach folgender Vorschrift: 1. Jede öffnende Klammer erhält eine um 1 höhere SCHACHTELUNGSTIEFE, wenn die zuletzt markierte Klammer (= Vorgängerklammer) eine öffnende Klammer ist, andernfalls erhält sie die SCHACHTELUNGSTIEFE der Vorgängerklammer. 2. Jede schließende Klammer erhält eine um 1 kleinere SCHACHTELUNGSTIEFE, wenn die zuletzt markierte Klammer (= Vorgängerklammer) eine schließende Klammer ist, andernfalls erhält sie die SCHACHTELUNGSTIEFE der Vorgängerklammer.
3.	**Ergebnis feststellen** 1. Hat eine schließende Klammer die SCHACHTELUNGSTIEFE < 1, dann liegt keine ordnungsgemäße Liste vor. Es fehlen öffnende Klammern. 2. Hat eine schließende Klammer die SCHACHTELUNGSTIEFE $= 1$, dann markiert diese das Ende einer Liste. 3. Haben nur die erste und letzte Klammer die SCHACHTELUNGSTIEFE $= 1$ und ist diese sonst > 1, dann bilden alle Klammern eine Liste.

Tabelle 1.2: Verfahren zum Erkennen der Schachtelungstiefe bei Listen

```
  ( (HIER (SIND (VIELE)  (SCHOENE) ) )
  1 2     3     4     4  4       4 3 2

 ( (KLAMMERN)  (  (  (ZU)  ) ) ( (ZAEHLEN) ) ) )
 2 3         3  3  4  5  5  4 3 3 4       4 3 2 1

Ergebnis des Verfahrens:
  Es liegt eine ordnungsgemäße Liste vor!
```

Die zunächst ungewohnte Vielzahl von Klammern zeigt nach kurzer Eingewöhnungszeit ihre Vorteile. Wir sparen damit das Schreiben vieler `Begin-End`- oder `do-od`-Paare. Auch das Problem der Reichweite bei der Schachtelung von Alternativen, das sogenannte „baumelnde“ `ELSE`-Problem, wird vermieden.



1.1.2 Semantik: Regeln zum Auswerten (Evaluieren)

Der Prozess, betont im Namen *LISt Processing*, bewirkt die Auswertung von symbolischen Ausdrücken. Ein LISP-System liest einen symbolischen Ausdruck ein, z. B. von einer öffnenden Klammer bis zur korrespondierenden schließenden Klammer, und wertet diesen Ausdruck aus. Als Vollzug der Auswertung gibt es einen symbolischen Ausdruck zurück, der als Wert (engl.: *value*) des eingelesenen Ausdruckes betrachtet wird. Der Prozess besteht aus drei Phasen:

1. Einlesen (*READ-Phase*),
2. Auswerten (*EVAL-Phase*) und
3. Ausgeben (*PRINT-Phase*).

Ein LISP-System kann nach diesen drei Phasen den nächsten symbolischen Ausdruck auswerten. Der Prozess ist somit ein *READ-EVAL-PRINT*-Zyklus (↪ Abbildung 1.4 S. 21). In diesem Buch sind die drei Prozessphasen durch folgende Notation abgegrenzt:

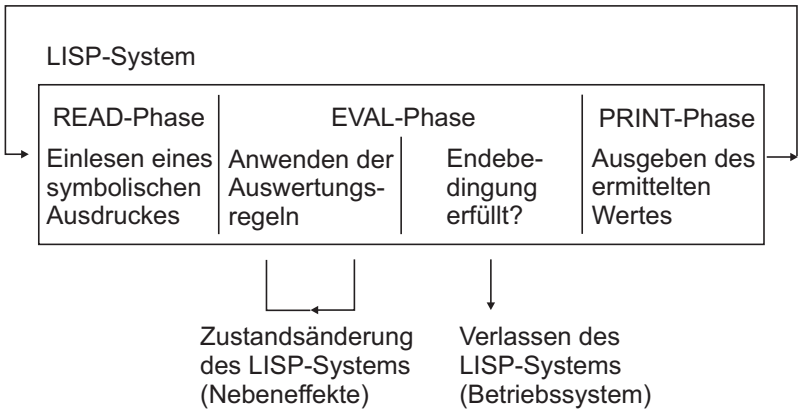
```
eval> <sexpr> ==> <sexpr> ;Kommentar
```

Beispiel:

```
eval> (Abbuchung! Meyer-AG 67.23) ==> 41.45
                                  ;Neuer Saldo
```

Der nach dem Prompt `eval>` notierte symbolische Ausdruck durchläuft die *READ*- und die *EVAL*-Phase. Der symbolische Ausdruck, notiert nach dem Pfeil `==>`, ist der ausgegebene Wert, d. h. das Ergebnis der *PRINT*-Phase. Der Text in einer Zeile, notiert hinter einem Semikolon, ist zusätzlicher Kommentar.

Die Regeln zur Auswertung („*Evaluierung*“) definieren die operationale Semantik von LISP, d. h. die Bedeutung der symbolischen Ausdrücke für die Abarbeitung durch das LISP-System. Diese *mechanische* Semantik ist in dem *eval*-Konstrukt des LISP-Systems abgebildet. Die Regeln sind daher selbst in LISP formuliert.



Legende:
Der *READ-EVAL-PRINT*-Zyklus wird häufig als *READ-EVAL-PRINT*-Loop (kurz: *REPL*) bezeichnet (↪ z. B. [186]).

Abbildung 1.4: Der LISP-Prozess: *READ-EVAL-PRINT*-Zyklus

Das Evaluieren stützt sich auf vier Basisregeln:

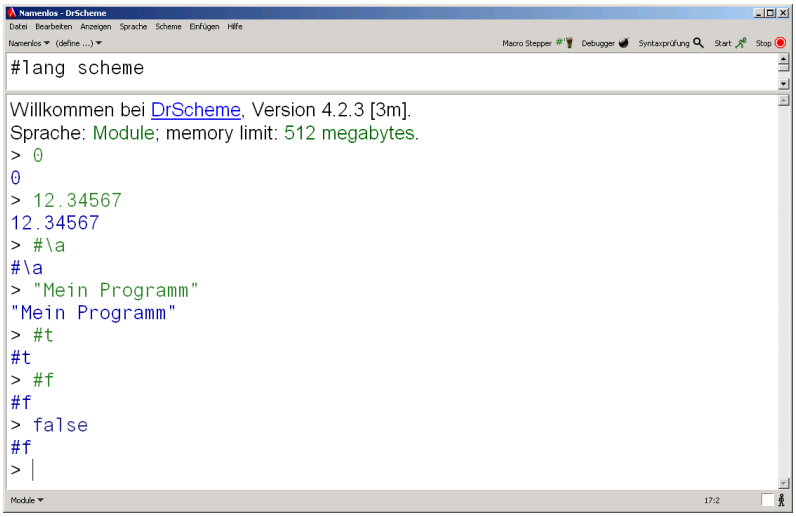
- *EVAL*-Regel 1: Evaluieren von Zahlen, Zeichen, Zeichenketten und den Wahrheitswerten `true` bzw. `false`

Ist der zu evaluierende symbolische Ausdruck eine Zahl, ein Zeichen (engl.: *character*), oder eine Zeichenkette (engl.: *string*), dann ist sein Wert der symbolische Ausdruck selbst. Liegt ein Wahrheitswert „wahr“ (`true` oder `#t`) oder „nicht wahr“ (`false` oder `#f`) vor, dann ist der Wert der Wahrheitswert selbst, allerdings abgebildet in der Standardrepräsentation des jeweiligen LISP-Systems.[4]

Beispiele zur *EVAL*-Regel 1 (↪ Abbildung 1.5 S. 22)

```
eval> 0 ==> 0
eval> 12.34567 ==> 12.34567
eval> #\a ==> #\a  ;Zeichen (engl.: character)
eval> "Mein Programm: "
        ==> "Mein Programm: "
           ;Zeichenkette
           ; (engl.: string)

eval> #t  ==> true ;Repräsentation
```

[4] Hinweis: Bei einigen LISP-System, z. B. bei PC Scheme (↪ `http://www.cs.indiana.edu/scheme-repository/imp.html` (Zugriff: 25-Oct-2009)) werden im Falle einer Entscheidung neben `true` alle Werte ungleich `NIL` bzw. `nil` als „wahr“ aufgefasst.



Legende:
Dr. Scheme ↪`http://www.plt-scheme.org/` (Zugriff: 24-Oct-2009)

Abbildung 1.5: Beispiele zur *EVAL*-Regel 1

```
                    ; des Wahrheits-
                    ; wertes "wahr"

eval> false ==> false ;Repräsentation
                    ; des Wahrheitswertes
                    ; "nicht wahr"
```

- *EVAL-Regel 2:* Evaluieren einer Liste des Typs
  $(< funktion >< arg_1 > \ldots < arg_n >)$

Ist der symbolische Ausdruck eine Liste des Typs $(< funktion >< arg_1 > \ldots < arg_n >)$ dann ergibt sich der Wert durch Evaluieren der Argumente $< arg_1 > \ldots < arg_n >$ und anschließender Anwendung der Funktion auf diese Werte. Hier sei eine Funktion durch ein Symbol benannt. Z. B. benennt das Symbol $+$ üblicherweise die Additionsfunktion. (Anonyme Funktionen ↪ Abschnitt 1.1.4 S. 35).

Hinweis: Begriff *Funktion*.
Wie in LISP-ischer Terminologie üblich (↪ [2, 178]) sprechen wir hier von einer Funktion oder auch LISP-Funktion, obwohl das Konstrukt im mathematischen Sinne keine Funktion darstellt, sondern eine Prozedur oder einen Algorithmus. Der Begriff *Prozedur* betont den Prozess für die Berechnung von Werten für eine Funktion. Mathematisch ist eine Funktion eine existierende eindeutige Zuordnung von Elementen der Definitionsmenge (engl.: *domain*) zu Elementen der Wertemenge (engl.: *range*),

wobei es keinen Berechnungsprozess gibt. Wenn verkürzt formuliert ist: „Eine Funktion F gibt den Wert Y zurück.“, dann ist damit gemeint, dass die algorithmische Repräsentation der Funktion F den Wert Y produziert; kurz: Es handelt sich um die Ausführung der Prozedur F.

In den Anfängen der Informatik wurde die Funktion oft wie bei der *Registertransfersprache* (RTS) definiert: „*Funktionen sind Abkürzungen für häufiger auftretende und/oder komplizierter Boolsche-Verknüpfungen. Sie sind wie die Variablen vom binären Typ. Die Eingangsgrößen von Funktionen sind entweder Ausdrücke oder dezimale Konstanten (Indices).*“ (↪ [141])

Mit der Nennung des Operations-Symbols als erstes Element einer Liste folgt LISP der *Präfix-Notation*. Die Präfix-Notation, auch Polnische Notation nach der polnischen Logikerschule um *Jan Łukasiewicz*[5] benannt, bezieht sich in LISP auf alle Operationen. Arithmetische Funktionen wie Addition, Subtraktion, Multiplikation oder Division haben damit eine gegenüber der Schulmathematik ungewohnte Schreibweise. Die Präfix-Notation gewährleistet eine einheitliche Syntax, unabhängig von der Zahl der Argumente. Die Operation steht stets an erster Stelle in einer Liste.

Beispiele zur *EVAL*-Regel 2 (↪ Abbildung 1.6 S. 24)

```
eval> (* 11111111111111111111111111111111111111111111111111111111
         22222222222222222222222222222222222222222222222222222222
         33333333333333333333333333333333333333333333333333333333)

     ==> 82304526748971193415637860082304526748971193390946502
         05761316872427983539094650205761316872674897119341563
         78600823045267489711934156378 6
eval> (sin 0) ==> 0
eval> (cos 0.5) ==> 0.8775825618903728
```

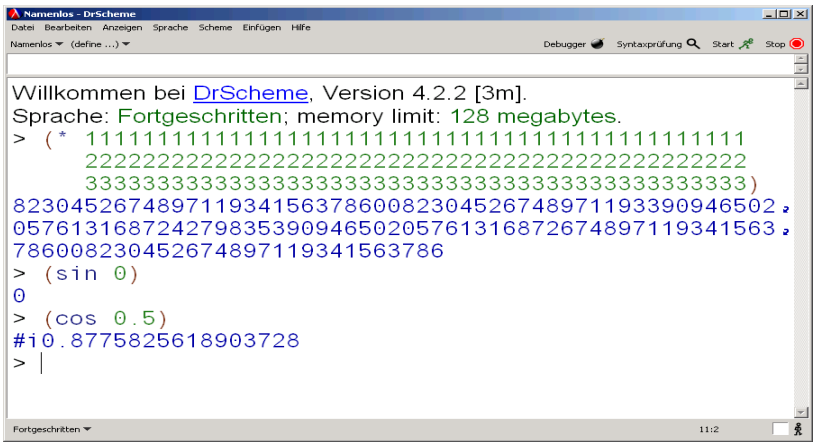
Die Argumente $< arg_1 > \ldots < arg_n >$ können Listen sein und damit ebenfalls dem Konstrukttyp $(< funktion >< arg_1 > \ldots < arg_n >)$ entsprechen, so dass auch für sie die *EVAL*-Regel 2 anzuwenden ist.

```
eval> (+ 1 (+ 1 3) 1) ==> 6
eval> (max (+ 0.4 0.6) (+ 2.34 1)) ==> 3.34
eval> (+ (- (+ (- 1 1) 1) 1) 1) ==> 1
```

- *EVAL*-Regel 3: Evaluieren einer Liste des Typs
  $(< spezielles - symbol >< arg_1 > \ldots < arg_n >)$

Ist der symbolische Ausdruck eine Liste des Typs
$(< spezielles - symbol >< arg_1 > \ldots < arg_n >)$ dann hängt der Wert von der Definition des speziellen Symbols ab. Ob von den einzelnen Argumenten $< arg_1 > \ldots < arg_n >$ einige evaluiert werden oder nicht, bestimmt die Definition, die das spezielle Symbol benennt. Ein LISP-System

[5] *J. Łukasiewicz*, ⋆ 21.12.1878 in Lemberg, † 13.02.1956 in Dublin, war nicht nur Mathematiker (Logiker), sondern auch Philosoph.



Legende:
Dr. Scheme ↪`http://www.plt-scheme.org/` (Zugriff: 24-Oct-2009)

Abbildung 1.6: Beispiele zur *EVAL*-Regel 2

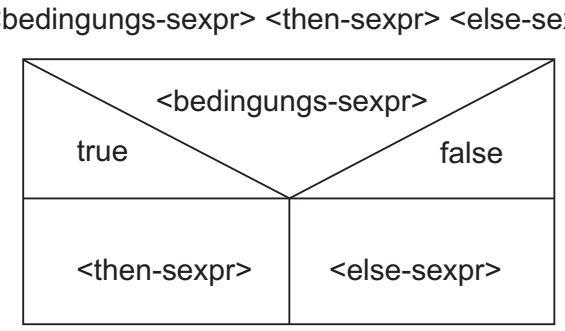
erkennt seine speziellen Symbole, so dass es nicht die *EVAL*-Regel 2 anwendet, sondern die für das spezielle Symbol definierte Evaluierung ausführt.

Beispiele zur *EVAL*-Regel 3 (↪ Abbildung 1.9 S. 27)

Spezielles Symbol `if` Bei der Alternative (↪ Abbildung 1.7 S. 25) in der Art „*if...then...else...end-if*“, die anhand des speziellen Symbols `if` erkannt wird, evaluiert das LISP-System zunächst das erste Argument, hier `<bedingungs-sexpr>`. Sein Wert bestimmt die weitere Abarbeitung, d. h. welches Argument als nächstes zu evaluieren ist. Ist der Wert von `<bedingungs-sexpr>` gleich `true` oder `#t`, d. h. die Bedingung gilt als erfüllt („True-Fall“), dann wird `<then-sexpr>` evaluiert und der gesamte if-Ausdruck hat den Wert von `<then-sexpr>`.

Ist der Wert von `<bedingungs-sexpr>` gleich `false` oder `#f`, d. h. die Bedingung gilt als nicht erfüllt („False-Fall“), wird die Alternative, formuliert als drittes Argument, evaluiert. Im „False-Fall“hat der gesamte `if`-Ausdruck den Wert von `<else-sexpr>`.

```
eval> (if false "Ja" "Nein") ==> "Nein"
eval> (if #t "Ja" "Nein") ==> "Ja"
eval> (define Antwort? #t)
eval> (if Antwort?
        "Dank für die Zustimmung."
        "Denken Sie mit!")
```



Legende:
Grafische Darstellung nach DIN 66261 (Struktogramme ↪ [138])

Abbildung 1.7: Konstrukt Alternative „*if…then…else…end-if*

```
==> "Dank für die Zustimmung."
;Bei diesem Rückgabewert hat
; das Symbol Antwort? den Wert true.
```

Hinweis: Was ist „wahr“?
Klassisch ist in LISP-Systemen alles wahr, was ungleich `false` oder `#f` ist, z. B. auch die Zahl 7 oder die Zeichenkette `"Alles klar?"`. So auch in *PLT-Scheme (DrScheme)* bei der Einstellung `#lang scheme`, also in der Sprachwahl „Module“ (↪ Abbildung 1.8 S. 26).

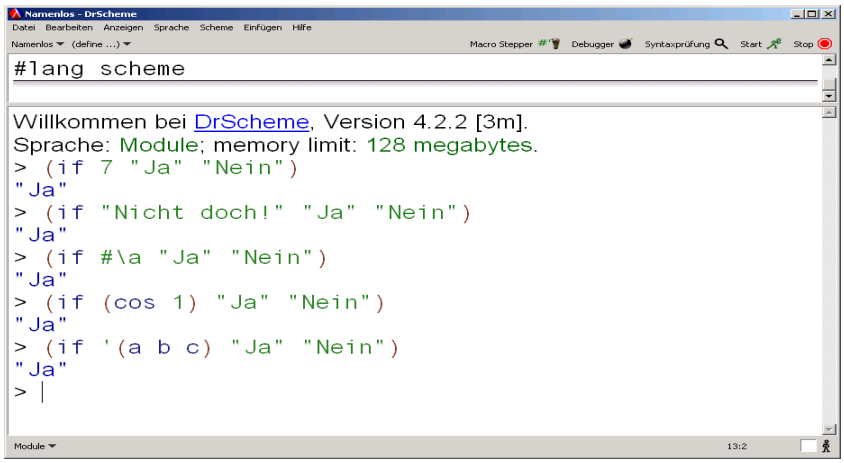
```
eval> (if 7 "Ja" "Nein") ==> "Ja"
> (if "Nicht doch!" "Ja" "Nein") ==> "Ja"
> (if #\a "Ja" "Nein") ==> "Ja"
> (if (cos 1) "Ja" "Nein") ==> "Ja"
> (if '(a b c) "Ja" "Nein") ==> "Ja"
```

Spezielles Symbol `define`

`eval> (define` < *symbol* > < *sexpr* >)

Das spezielle Symbol `define` bindet ein Symbol an den Wert von < *sexpr* > im aktuellen Kontext, d. h. die Bindung findet in der Umgebung statt, in dem die `define`-Auswertung erfolgt. Dabei wird das erste Argument, hier < *symbol* >, nicht evaluiert; das zweite, hier `<sexpr>`, jedoch. Der Wert des gesamten `define`-Konstruktes ist uninteressant und deshalb prinzipiell auch nicht spezifiziert.[6]

[6] Hinweis: Bei einigen LISP-System, z. B. bei *PC Scheme* wird im Falle des `define`-Konstruktes das Symbol zurückgegeben. Dies erfolgt, weil der *READ-EVAL-PRINT*-Zyklus für jeden eingelesenen symbolischen Ausdruck einen Wert in der *PRINT*-Phase zurückgibt.



Legende:
Dr. Scheme ↪`http://www.plt-scheme.org/` (Zugriff: 24-Oct-2009)

Abbildung 1.8: Beispiele zu „wahr“

Die Aufgabe des `define`-Konstruktes ist nicht die Rückgabe eines Wertes, sondern das Binden eines Symbols an einen Wert. Anders formuliert: Zwischen dem Symbol und seinem Wert entsteht eine Verbindung (*Assoziation*). Das Symbol „benennt“ den Wert. Das `define`-Konstrukt ist somit der „Benennungsoperator“ für symbolische Ausdrücke.

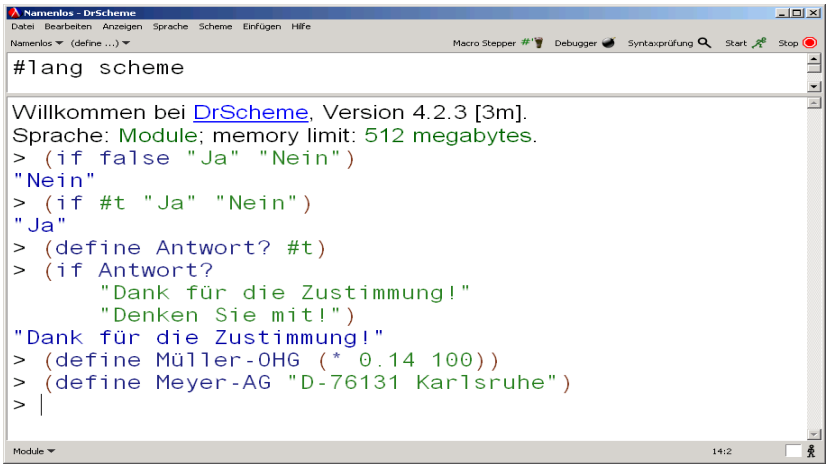
- *Common LISP*: `setq`-Konstrukt.
  Die Bindung übernimmt in Common LISP und in vielen anderen Dialekten das `setq`-Konstrukt. Sein (Rückgabe-)Wert ist nicht das Symbol, sondern der Wert, an den das Symbol gebunden wird.

Beispiele zum `define`-Konstrukt:

```
eval> (define Müller-OHG (* 0.14 100))
        ==> ;kein Rückgabewert
eval> (define Meyer-AG "D-76131 Karlsruhe")
        ==> ;kein Rückgabewert

[Common LISP:
C-eval> (SETQ MÜLLER-OHG (* 0.14 100)) ==> 14 ]
```

- *EVAL*-Regel 4: Evaluieren eines Symbols

Ist der symbolische Ausdruck ein Symbol, dann ist sein Wert derjenige symbolische Ausdruck, an den es in der (aktuellen) Umgebung



Legende:
Dr. Scheme ↪`http://www.plt-scheme.org/` (Zugriff: 24-Oct-2009)

Abbildung 1.9: Beispiele zur *EVAL*-Regel 3

gebunden ist. Gibt es keine entsprechende Bindung, dann führt das Evaluieren des Symbols zu keinem Wert, sondern abhängig vom jeweiligen LISP-System zu einer Fehlermeldung.[7]

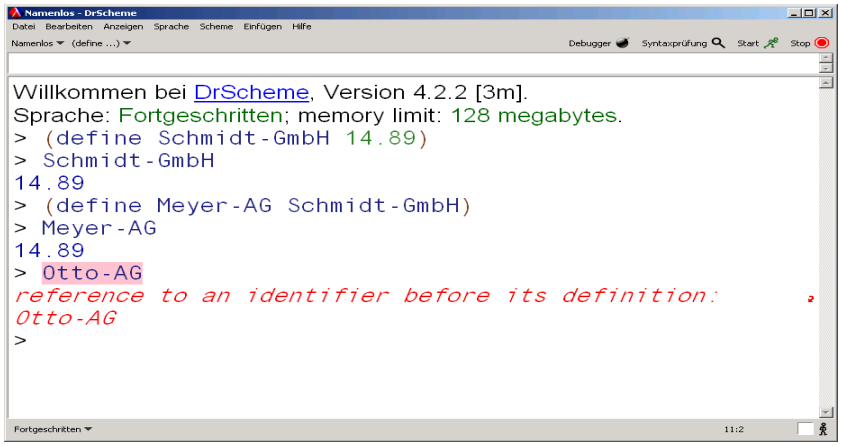
Beispiele zur *EVAL*-Regel 4 (↪ Abbildung 1.10 S. 28)

```
eval>(define Schmidt-GmbH 14.89)
eval> Schmidt-GmbH ==> 14.89
eval>(define Meyer-AG Schmidt-GmbH)
eval> Meyer-AG ==> 14.89
eval> Otto-AG
      ==> ERROR ...
     ;reference to an identifier
     ; before its definition: Otto-AG
```

1.1.3 Repräsentation von Daten und Programm

Im Mittelpunkt der Softwarekonstruktion steht der Begriff *Algorithmus*. Darunter versteht man ein eindeutig bestimmtes Verfahren (Prozess)

[7] Hinweis: Bei einigen LISP-System, z. B. bei *PC Scheme*,
(↪ `http://www.cs.indiana.edu/scheme-repository/imp.html`
(Zugriff: 25-Oct-2009)), erfolgt dann ein Übergang in einen Fehlerkorrekturmodus (engl.: *inspect/debug-level*).



Legende:
Dr. Scheme ↪`http://www.plt-scheme.org/` (Zugriff: 24-Oct-2009)

Abbildung 1.10: Beispiele zur *EVAL*-Regel 4

oder die Beschreibung eines solchen Verfahrens. Diese intuitive Definition verdeutlicht, dass es bei einem Algorithmus nicht nur um ein Verfahren geht, sondern auch um seine Beschreibung. Sie kann in einer natürlichen oder formalen Sprache (z. B. in LISP) oder mit Hilfe eines grafischen Darstellungsmittels (z. B. Struktogramm) erfolgen. Bedeutsam ist die Eindeutigkeit der Beschreibung mit Hilfe eines endlichen Textes. Bestimmt wird ein Algorithmus durch die Vorgabe:

1. einer Menge von Eingabe-, Zwischen- und Endgrößen,
2. einer Menge von Elementaroperationen und
3. der Vorschrift, in welcher Reihenfolge welche Operationen wann auszuführen sind.

Exkurs: Begriff *Algorithmus*
Bei den theoretisch fundierten Definitionen des Alogrithmusbegriffs z. B. von *A. M. Turing* oder *A. A. Markow* geht es um Aufzählbarkeit, Berechenbarkeit und Entscheidbarkeit. In diesem Zusammenhang hat *Alonzo Church* nachgewiesen, dass bei jeder möglichen Präzisierung des Algorithmusbegriffs die dadurch entstehende Klasse von berechenbaren Funktionen mit der Klasse der allgemein rekursiven Funktionen zusammenfällt (Näheres ↪ z. B. [109]).

Im heutigen Programmieralltag z. B. bei der Erstellung eines Buchungsprogrammes in *Java*[8] spricht man eher vom *Programm* als vom *Al-*

[8] *Java* ist eine objekt-geprägte Programmiersprache und eingetragenes Warenzeichen der Firma *Sun Microsystems*.

gorithmus. Dabei unterscheidet man strikt zwischen den *Daten* (Eingabe-, Zwischen- und Endgrößen) und dem *Programm* (Vorschrift basierend auf Elementaroperationen), obwohl im *von Neumann*-Rechner die Trennungslinie zwischen Programm und Daten unscharf ist. Man verbindet mit dem Begriff *Programm* etwas Aktives wie Daten erzeugen, modifizieren oder löschen. Die *Daten* selbst sind dabei passiv. Sie werden vom *Programm* manipuliert. Diese Unterscheidung hat für symbolische Ausdrücke wenig Aussagekraft; denn jeder symbolische Ausdruck ist einerseits ein *Programm*, dessen Ausgabe (Wert) er im Zusammenwirken mit den *EVAL*-Regeln definiert, und andererseits das Mittel, um in LISP *Daten* darzustellen.

Von Bedeutung ist in LISP die Frage, ob und wie ein symbolischer Ausdruck evaluiert wird. Sie ist entscheidend, wenn man eine Unterteilung in *Daten* und *Programm* vornehmen will. Bei einer Zeichenkette, z. B.:
`"Programmieren ist das Abbilden von Wissen"`,
besteht kein Grund diese zu evaluieren, weil sie selbst das Ergebnis ist. Zahlen, Zeichen und Zeichenketten haben eher passiven Charakter; sie zählen zur Kategorie *Daten*. Die Liste ist ambivalent. Sie gehört sowohl zur Kategorie *Daten* als auch zur Kategorie *Programm*. Das folgende Beispiel verdeutlicht diese Ambivalenz.

Beispiel: Buchungsprogramm
Vom Konto `Meyer-AG` ist ein Betrag von €67,23 abzubuchen. Wir formulieren folgende Liste:

```
eval> (Abbuchen! Meyer-AG 67.23) ==> 1200.45
```

Wir erhalten den neuen Saldo €1200.45, vorausgesetzt es existiert eine Funktion `Abbuchen!` und ein Konto `Meyer-AG` mit dem alten Saldo €1267.68. Das Ergebnis von 1200.45 entstand durch „Laufenlassen“ des Programmes:
`(Abbuchen! Meyer-AG 67.23)`

↪ `http://java.com/de/download/index.jsp` (Zugriff: 1-Feb-2010)

Diese Liste führt zu einer Ausgabe (Output), indem sie dem *READ-EVAL-PRINT*-Zyklus übergeben wird; wir müssten sie der Kategorie *Programm* zuordnen.

Jetzt wollen wir annehmen, dass die Information, vom Konto `Meyer-AG` sei ein Betrag von €67.23 abzubuchen, im Sinne von Daten einem LISP-System einzugeben ist. Wir können z. B. unterstellen, es sei nur zu beschreiben, dass abzubuchen ist, aber die Buchung selbst ist nicht zu vollziehen, weil das Konto `Meyer-AG` noch nicht freigegeben ist. Zunächst könnte diese Information statt als Liste als Zeichenkette dem *READ-EVAL-PRINT*-Zyklus übergeben werden:

```
eval> "Abbuchen! Meyer-AG 67.23"
         ==> "Abbuchen! Meyer-AG 67.23"
```

Behalten wir jedoch die Liste bei, dann dürfen die *EVAL*-Regeln auf diese Liste nicht angewendet werden. Die Auswertung wäre zu blockieren, so dass `Abbuchen!` nicht als Name einer Funktion interpretiert wird. *EVAL*-Regel 3 bietet die Möglichkeit, anhand eines speziellen Symbols über die Auswertung der Argumente zu entscheiden. Das spezielle Symbol `quote` repräsentiert diesen „Auswertungsblocker". Um *EVAL*-Regel 3 zur Anwendung zu bringen, muss die Liste als Argument für das spezielle Symbol `quote` geschrieben werden. Wir formulieren daher:

```
eval> (quote (Abbuchen! Meyer-AG 67.23))
         ==> (Abbuchen! Meyer-AG 67.23)
```

Da das Einfügen eines symbolischen Ausdrucks in die Liste mit dem Auswertungsblocker `quote` als erstem Element häufig vorkommt, gibt es dafür eine Kurzschreibweise mit dem Hochkomma `'` (engl.: quote mark). Es wird in der *READ*-Phase zum eigentlichen `QUOTE`-Konstrukt aufgelöst.

```
eval> '(Abbuchen! Meyer-AG 67.23)
         ==> (Abbuchen! Meyer-AG 67.23)
```

Wird das Evaluieren einer Liste blockiert, dann verhält sie sich analog zu einer Zeichenkette. Sie bleibt „bestehen". Wir müssten sie aus dieser Sicht der Kategorie *Daten* zuordnen.

Einprägsam formuliert: Die *Quotierung* blockiert die Auswertung. Ein quotierter symbolischer Ausdruck entspricht einer Konstanten.

Beispiele zum `quote`-Konstrukt:

```
eval> '(A B C ) ==> (A B C)

eval> ('A 'B 'C) ==> ERROR ...
         ;procedure application:
         ; expected procedure, given:
         ; 'A; arguments were: 'B 'C
         ; Nach EVAL-Regel 3 ist der Wert von
```

```
          ; (quote A) ==> A
          ; Dieses  A ist weder der Programmcode
          ; einer Funktion entsprechend EVAL-Regel 2
          ; noch der eines speziellen Symbols ent-
          ; sprechend EVAL-Regel 3.

eval> (sin '(+ 0.2 0.3)) ==> ERROR ...
         ;sin: expects argument of type
         ; <number>; given (list '+ 0.2 0.3)
         ; Die Quotierung des Arguments verhindert
         ; die Auswertung von (+ 0.2 0.3) ==> 0.5.
         ; Der Sinus einer Liste ist nicht defi-
         ; niert, so dass eine Fehlermeldung erfolgt.

eval> (sin (+ 0.2 0.3)) ==> 0.479425538604203
```

Hinweis: *Flüchtigkeitsfehler*
Häufig haben (Flüchtigkeits-)Fehler ihre Ursache in einer vergessenen oder falschen Hochkomma-Notierung. Prüfen Sie deshalb zunächst, ob der symbolische Ausdruck selbst oder sein Wert gemeint ist (↪ Abschnitt 1.2.4 S. 105).

Das Komplement zum `quote`-Konstrukt ist das `eval`-Konstrukt. Es bewirkt die Anwendung der EVAL-Regeln. Diese transformieren eine „Konstante“ als Datum wieder in die Rolle eines *Programms*. In LISP sind *Programme* und *Daten* gleichermaßen als symbolischer Ausdruck abgebildet. Als Merksatz formuliert: Die Repräsentation von LISP-Programmen als LISP-Daten ist eine Haupteigenschaft dieser Programmiersprache. Sie ermöglicht das Manipulieren von Programmen durch Programme ohne die übliche Trennung in Systemprogrammierung und Anwendungsprogrammierung. Jeder LISP-Benutzer kann sein LISP-System selbst verbessern, wobei viele dieser Verbesserungen letztlich zur Verbesserung der kaufbaren LISP-Systeme geführt haben (↪ [123]).

Die folgenden `eval`-Beispiele dienen als erster Einstieg. Im Zusammenhang mit der Diskussion des Umgebungskonzeptes wird das EVAL-Konstrukt weiter erklärt.

Beispiele zum `eval`-Konstrukt:

```
eval> '(+ 1 2) ==> (+ 1 2)
eval> (eval '(+ 1 2)) ==> 3
eval> (eval '(eval '(+ 1 2))) ==> 3

eval> (define Meine-Addition '(+ 1 2))
eval>  Meine-Addition ==>  (+ 1 2)
eval> (eval Meine-Addition) ==> 3

eval> (define Nachricht
```

```
        '(if Antwort? "Meyer AG zahlt!"
                      "Meyer AG zahlt nicht!"))
eval> (define Antwort? #f)
eval> (eval Nachricht) ==> "Meyer AG zahlt nicht!"
eval> (define Anwort? 'Ja)
eval> (eval Nachricht) ==> "Meyer AG zahlt!"
```

1.1.4 Konstrukt: Schnittstelle und Implementation

Bisher wurde gezeigt: Ist der auszuwertende symbolische Ausdruck eine Liste, dann definiert das erste Element, ob die restlichen Listenelemente (Argumente) evaluiert werden oder nicht. Die restlichen Listenelemente können als Eingabe aufgefasst werden. Sie sind Eingabe (*input*) für die Funktion, die durch das erste Listenelement benannt wird. Entsprechendes gilt, wenn das erste Element einen speziellen Fall benennt (*EVAL*-Regel 3).

Die Eingabeschnittstelle zum Programmcode eines Konstruktes, das angewendet wird, bestimmt die Zulässigkeit eines Argumentes. So muss die Anzahl der angegebenen Argumente der jeweiligen Schnittstellendefinition genügen. Zusätzlich muss der Typ eines Arguments passen. Arithmetische Funktionen fordern z. B. numerische Argumente, also symbolische Ausdrücke vom Typ Zahl bzw. Ausdrücke, die zum Typ Zahl evaluieren. Allgemein formuliert: Die Argumente müssen konsistent zur Schnittstelle sein.

Beispiele zur Anzahl und zum Typ von Argumenten:

```
eval> (if Antwort? "gestern" "heute" "morgen")
        ==> ERROR ... ;Zu viele Argumente
eval> (define ) ==> ERROR ...
                      ;Zu wenige Argumente
eval> (define Otto "Witzig" "Hudelig")
        ==> ERROR ... ;Zu viele Argumente

eval> (define "Meyer-AG" "Saldo ist EUR 123.45")
        ==> ERROR ... ;Wertassoziation verlangt
                      ; ein Symbol

eval> (define Meyer-AG "Saldo ist EUR 123.45")

eval> Meyer-AG  ==>
        "Saldo ist EUR 123.45"
        ;Zeigt den Wert der
        ; vorhergehenden define-Anwendung

eval> (+ 1 2 'Alter-Wert)
         ==> ERROR ...
```

```
;Nichtnumerischer Operand für eine
; arithmetische Operation
```

Betrachten wir einen symbolischen Ausdruck als gegebenes Konstrukt (als einen bereitgestellten Baustein) zur Abarbeitung einer spezifizierten Aufgabe, dann werden Angaben benötigt über:

1. die (Eingabe-)Schnittstelle,
2. den (Ausgabe-)Wert und über
3. die Nebeneffekte.

Hinweis: *Mehrere(Ausgabe-)Werte (engl.: multiple values)*
Manche LISP-Systeme, z. B. Common LISP, gestatten als Ergebnis des evaluierens die Rückgabe von mehreren Werten. Da mehrere Werte (hilfsweise) in einer Liste zusammenfassbar sind, betrachten wird diese Option nicht weiter.

zu 1) *Schnittstelle:* Die Kenntnis der Schnittstelle ist für die Wahl korrekter Argumente notwendig.

zu 2) *Wert:* Zumindest muss der Typ des symbolischen Ausdrucks bekannt sein, den wir als Wert des evaluierens erwarten, wenn wir diesen Wert als Argument für ein weiteres Konstrukt einsetzen wollen. Bekannt sein muss z. B., ob das Konstrukt eine Zahl, ein Symbol oder eine Liste als Wert hat.

Beispiel für einen falschen Argument-Typ:

```
eval> (+ 3 (define Meyer-AG 245.34) 4)
      ==> ERROR ...
;define: not allowed in an expression context...
;Die Additionsfunktion erfordert
; numerische Argumente. Kein
; Rückgabewert des Konstruktes
; (define Meyer-AG 245.34). Es kommt wegen
; der Inkompatibilität mit
; der Schnittstelle der Additionsfunktion
; zum Fehler.
```

zu 3) *Nebeneffekt:* Wenn symbolische Ausdrücke evaluiert werden, können sie nicht nur ihren Wert zurückgeben, sondern zusätzlich den Wert symbolischer Ausdrücke verändern. Sie verursachen einen Nebeneffekt, auch Seiteneffekt genannt. Bisher haben wir als einen solchen symbolischen Ausdruck das `define`-Konstrukt vorgestellt, d. h. die Liste mit dem speziellen Symbol `define` als erstem Element. Die Nebeneffekte eines Konstruktes müssen bekannt sein, um ihre Wertveränderungen einkalkulieren zu können.

Beispiel für einen Nebeneffekt:

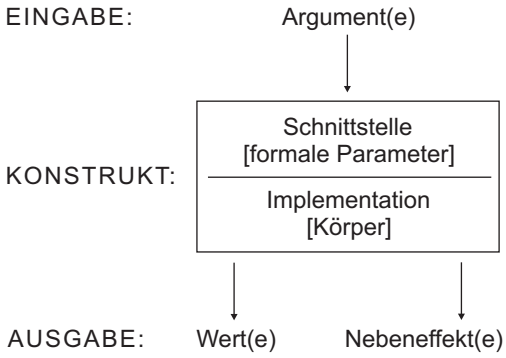


Abbildung 1.11: LISP-Konstrukt

```
eval> (define Schulze-GmbH 3.45)
eval> (define Meyer-AG
        (lambda ()
          (set! Schulze-GmbH 123.45)
          1000.00))
eval> (Meyer-AG)
      1000.0
eval> Schulze-GmbH
      123.45
      ;Durch Anwendung der selbst
      ; definierten Funktion Meyer-AG
      ; wurde der Wert von
      ; Schulze-GmbH modifiziert.
```

Exkurs: *Signatur*
Zur Spezifikation mit Hilfe von elementaren Funktionen (Operationen, nebeneffektfreien Konstrukten) fassen wir die Informationen:

1. Name der Funktion,
2. Typ der Argumente (*domain* der Funktion) und
3. Rückgabewert (*range* der Funktion)

unter dem Begriff „Signatur“ zusammen.

Solange es um Konstrukte geht, die das jeweilige LISP-System als eingebaute Konstrukte dem Benutzer direkt bereitstellt, ist ihre Implementation eine zweitrangige Frage. Bedeutsam sind für uns ihre Leistungskriterien: Anwortzeit- und Speicherplatz-Bedarf. Eine Darstellung des eigentlichen Programmcodes, der zur Anwendung kommt, hilft selten. Wir interessieren uns nicht für eine „externe“ maschinenabhängige Repräsentation des Programmcodes (wohlmöglich, in Form von hexadezimalen Zahlen). Wesentlich ist die Abstraktion in Form des Symbols, das

auf den Programmcode verweist (↪ Abbildung 1.11 S. 34). Beispielsweise interessiert uns bei der Additionsfunktion die Abstraktion in Form des + Symbols, verbunden mit Angaben zur Schnittstelle (hier: beliebig viele numerische Argumente), zum (Rückgabe-)Wert (hier: eine Zahl) und zur Frage von Nebeneffekten (hier: keine).

Soll ein selbst definiertes Konstrukt wie ein eingebautes Konstrukt benutzbar sein, dann ist es vorab zu implementieren. Implementation bedeutet hier: Es ist zu entscheiden, aus welchen dem LISP-System bekannten Konstrukten es zusammensetzbar ist. Unser Konstrukt besteht aus symbolischen Ausdrücken, auf die das LISP-System die *EVAL*-Regeln erfolgreich anwenden kann. Für die Einführung fassen wir nochmals zusammen:

Definition *Konstrukt:*
Ein Konstrukt („Baustein“) ist ein symbolischer Ausdruck, konzipiert für die Anwendung der *EVAL*-Regeln. Sein Einsatz setzt Informationen voraus über:

- die Schnittstelle (Anzahl der Argumente, Typ eines jeden Argumentes sowie die Festlegung, welche Argumente evaluiert werden),
- den (Rückgabe-)Wert (Vorhersage des Typs, z.B. ob Zahl, Symbol, Liste etc.) und über
- Nebeneffekte (Veränderung des eigenen Zustandes und/oder des Zustandes von anderen symbolischen Ausdrücken).

Hinweis: *Konstrukt*
Der lateinische Begriff *Konstrukt* (Construct, Constructum) bezeichnet eine Arbeitshypothese für die Beschreibung von Phänomenen, die der direkten Beobachtung nicht zugänglich sind, sondern nur aus anderen beobachteten Daten erschlossen werden können. In der Linguistik ist z. B. Kompetenz ein solches Konstrukt. Wir bezeichnen mit Konstrukt einen verwendbaren *Baustein*, der bestimmte Eigenschaften hat.

Beispiel zur Bezeichnung Konstrukt:
Wir sprechen von einem `define`-Konstrukt, wenn es sich um die folgende Zeile handelt:

```
(define foo '(interface value side-effects body))
```

`lambda`**-Konstrukt**

Im Rahmen der *EVAL*-Regeln wurde gezeigt (↪ Abschnitt 1.1.2 S. 20), dass sich der Wert eines Konstruktes durch die Applikation (Anwendung) des Programmcodes auf die Argumente ergibt. Dabei bestimmt

das erste Listenelement den Programmcode. Betrachten wir eine Funktion $y = f(x)$, dann ist die grundlegende Operation die Applikation von f auf ein Argument x, um den Wert y zu erhalten. Bisher wurden eingebaute Konstrukte (Funktionen), wie z. B. $+$ oder $-$ oder `max`, auf Argumente angewendet. Nun wollen wir eigene Funktionen konstruieren. Grundlage dazu bildet das `lambda`-Konstrukt. Es basiert auf dem λ-Kalkül, einem mathematischen Formalismus zur Beschreibung von Funktionen durch Rechenvorschriften.

Alonzo Church hat für die Präzisierung der Berechenbarkeit den λ-Kalkül formuliert ($\hookrightarrow$ [42]; zur Evaluierung von λ-Ausdrücken $\hookrightarrow$ [113]; zum Kalkülverstehen $\hookrightarrow$ [116]; zur Implementation $\hookrightarrow$ [2]). Programmiersprachen nach dem Modell des λ-Kalküls simulieren die sogenannten α- und β-Reduktionsregeln des Kalküls mit Hilfe einer Symbol-Wert-Tabelle. Die Simulation nutzt eine *Kopierregel*, wobei an die Stelle der formalen Parameter (λ-Variablen) im Original in der Kopie die aktuellen Parameter (Argumente) gesetzt werden. Der griechische Buchstabe λ selbst hat keine besondere Bedeutung. *Alonzo Church* hat ihn in Anlehnung an die Notation von *A. N. Whitehead* und *B. Russell* benutzt ($\hookrightarrow$ [116] p. 357). Zur Eingabe auf alphanumerischen Terminals wird üblicherweise das Symbol `lambda` verwendet.

Eine selbst definierte Funktion entsteht durch Evaluierung des `lambda`-Konstruktes.

`eval>` `(lambda` $<$ *schnittstelle* $><$ *rechenvorschrift* $>$`)`
`==>` $\#<$ *procedure* $>$

mit:

$<$ *schnittstelle* $>$	$\equiv$ Sie wird durch einen oder mehrere formale Parameter im Sinne von „Platzhalter-Variablen“ für die Argumente repräsentiert. Mehrere Platzhalter werden als Elemente einer Liste notiert. Die formalen Parameter bezeichnet man als `lambda`-Variablen oder als lokale Variablen. Die formalen Parameter werden durch Symbole angegeben.
$<$ *rechenvorschrift* $>$	$\equiv$ Sie wird durch Konstrukte, bei denen formale Parameter an Stellen von Argumenten stehen, konstruiert. Sie wird auch als Term (engl.: *form*) bezeichnet, um zu verdeutlichen, dass es ein symbolischer Ausdruck ist, der evaluiert wird. Im Zusammenhang mit der Schnittstelle bezeichnet man die Rechenvorschrift als (Funktions-)Körper (engl.: *body*).
$\#<$ *procedure* $>$	$\equiv$ Sie ist der Rückgabewert des *READ-EVAL-PRINT*-Zyklus und repräsentiert hilfsweise das entstandene Funktionsobjekt, d. h. den Programmcode, der durch die Schnittstelle und die Rechenvorschrift definiert ist.

Exkurs: Begriff *Argument*
Argument ist die Bezeichnung für *Element der Definitionsmenge* einer Funktion. Ist $f(x)$ definiert, so nennt man (salopp) auch die Variable x Argument. Exakt sind jedoch nur die für x einzusetzenden Werte als die Argumente von f zu bezeichnen (z. B. ↪ [161]).

Exkurs: Begriff *Variable*
In Programmiersprachen ist die Variable primär als Gegensatz zur Konstanten zu sehen. Sie ist mit einer *Quantität* verbunden, die variiert werden kann. In der Mathematik benutzt man den Begriff *Variable* bei Funktionen im Sinne eines bestimmten, aber anonymen Elements einer Definitionsmenge (z. B. „X sei eine nichtnegative rationale Zahl …“).

Beispiel für die Definition einer Funktion:
Es ist eine Funktion `Abbuchen` zu definieren, die von einem Konto einen Betrag abbucht. Um den Vorgang leicht verstehen zu können, sind einige Zwischenschritte angebracht. Als ersten Schritt formulieren wir folgendes `lambda`-Konstrukt:

```
eval> (lambda
        ;Schnittstelle
        (Alter_Saldo Abbuchungsbetrag)
           ; Rechenvorschrift
           (- Alter_Saldo Abbuchungsbetrag)
              ) ==> #<procedure>
```

Die Schnittstelle hat die beiden formalen Parameter `Alter_Saldo` und `Abbuchungsbetrag`. Die Rechenvorschrift besteht nur aus der Subtraktionsfunktion, so dass wir eine Zahl als (Rückgabe-)Wert erwarten, wenn die entstandene Funktion angewendet wird.

Wegen der Präfix-Notation bestimmt das erste Element einer Liste den anzuwendenden Programmcode. Beispielsweise benennt das Symbol Bindestrich „-“ das Funktionsobjekt (den Programmcode) für die Subtraktion. Ermittelt man den Wert vom Symbol Bindestrich (gemäß *EVAL*-Regel 4, ↪ Abschnitt 1.1.2 S. 20), dann erhält man das Funktionsobjekt und kann es auf die Werte seiner Argumente anwenden. Entsprechend ist das `lambda`-Konstrukt als erstes Element einer Liste zu notieren. Die restlichen Listenelemente sind die Argumente, auf die das durch Evaluierung erzeugte Funktionsobjekt angewendet wird.

Anwendung (Applikation) des `lambda`-Konstruktes:

```
eval> ((lambda < schnittstelle >
                 < rechenvorschrift >)
                   < arg_1 > ... < arg_n >)
                     ==>< wert >
```

Da die Schnittstellendefinition zu erfüllen ist, sind in unserem Beispiel zwei weitere Listenelemente als Argumente anzugeben.

```
eval>  ;;1. Listenelement ergibt Funktion
      ((lambda (Alter_Saldo Abbuchungsbetrag)
         (- Alter_Saldo Abbuchungsbetrag))
       110.00 ;2. Listenelement
              ; ist ihr 1. Argument
       50.00  ;3. Listenelement
              ; ist ihr 2. Argument
        ) ==> 60. ;Wert
```

Die Zuordnung des ersten Arguments, hier 110.00, zur `lambda`-Variablen `Alter_Saldo` und des zweiten Arguments, hier 50.00, zur `lambda`-Variablen `Abbuchungsbetrag` wird durch die Reihenfolge definiert. Die Argumente werden gemäß ihrer Reihenfolge an die `lambda`-Variablen gebunden. Die Bindung der `lambda`-Variablen folgt der Idee von Stellungsparametern. Ein gleichzeitiges Vertauschen der Reihenfolge der (nebeneffektfreien) Argumente und der `lambda`-Variablen führt daher zum gleichen Wert.

```
eval> ((lambda (Abbuchungsbetrag Alter_Saldo)
          (- Alter_Saldo Abbuchungsbetrag))
         50.00 110.00)
        ==> 60.
```

Für den Wert des Konstruktes ist die Benennung der `lambda`-Variablen unerheblich; soweit nicht Namenskonflikte mit freien Variablen auftreten (Näheres zur Konfliktlösung ↪ Abschnitt 2.4.4 S. 259). Für das Verstehen eines Konstruktes ist die Benennung jedoch von entscheidender Bedeutung (Näheres dazu ↪ Abschnitt 3.1.1 S. 377). Das folgende Konstrukt hat die Symbole `X` und `Y` als `lambda`-Variablen und bewirkt ebenfalls die obige Abbuchung.

```
eval> ((lambda (X Y) (- Y X))
        50.00 110.00)
        ==> 60. ;So nicht!
```

Wird das `lambda`-Konstrukt evaluiert, entsteht eine anonyme Funktion. Anonym, weil sie keinen Namen hat. Um die definierte Funktion zu benennen, wird ein Symbol an sie gebunden, d. h. der Wert eines Symbols ist das evaluierte `lambda`-Konstrukt. Eine solche Bindung bewirkt in Scheme das `define`-Konstrukt.

```
eval> (define Abbuchen
        (lambda (Alter_Saldo Abbuchungsbetrag)
          (- Alter_Saldo Abbuchungsbetrag))
      )
eval> Abbuchen ==> #<procedure:Abbuchen>
```

Hinweis: *Funktionsdefinition*
Das Konstrukt mit dem der Benutzer eine Funktion definieren kann, hat in LISP-Systemen verschiedene Namen und unterschiedliche Wirkungen. In *TLC-LISP* heißt es `DE`, in *Common LISP* `DEFUN`. Allen gemeinsam ist das Verknüpfen eines Symbols mit einem Funktionsobjekt.

Das Symbol `Abbuchen` kann nun wie ein eingebautes Konstrukt, z. B. wie das Subtraktions-Konstrukt, angewendet werden.

```
eval> (Abbuchen 110.00 50.00) ==> 60.
eval> (- 23.40 2.30) ==> 21.1
```

Als leicht merkbares, begrenzt verallgemeinerbares Motto formuliert: *Form follows function*. Dieser beinahe klassische Leitsatz der (Industrie-)-Designer gilt auch in LISP.

```
  (define Verdopplung (lambda (Zahl) (* 2 Zahl)))
            |           |           |        |
            +-function--+           +--form--+
```

Das Anwenden einer Funktion auf Argumente ist mit dem `apply`-Konstrukt explizit notierbar. Es entspricht z. B. :

```
eval> (< funktion > ' < arg_1 > ... ' < arg_n >)
          ==>< value >
eval> (apply< funktion > ' (< arg_1 > ... < arg_n >))
          ==> < value >
```

Wir können daher auch formulieren:

```
eval> ;;Applikation einer anonymen Funktion
      (apply
       (lambda
         (Alter_Saldo Abbuchungsbetrag)
         (- Alter_Saldo Abbuchungsbetrag))
       '(110.00 50.00))
        ==> 60.

eval> ;;Applikation der
      ;; benannten Funktion Abbuchen
      (apply Abbuchen '(110.00 50.00))
        ==> 60.
```

Das explizit notierbare Applikationskonstrukt hat folgende Struktur:

```
eval> (apply < funktion >< argumentenliste >)
         ==> < value >
```

Das `apply`-Konstrukt evaluiert seine beiden Argumente, d. h. sowohl den Ausdruck $<$ *funktion* $>$ als auch den Ausdruck $<$ *argumentenliste* $>$. Die Evaluierung von $<$ *argumentenliste* $>$ muss zu einer Liste führen. Die Elemente dieser Liste sind die Werte an welche die `lambda`-Variablen von $<$ *funktion* $>$ gebunden werden und zwar in der genannten Reihenfolge.

```
eval> (define Vorgang-1 '(110.00 50.00))
eval> (apply Abbuchen Vorgang-1) ==> 60.
```

Exkurs: *Mehrere Argumente*
Bei dem Ausdruck `(<f> <sexpr>)` sei `<f>` eine Funktion und der Wert von `<sexpr>` die Liste L. Ist `<f>` für ein Argument definiert, dann wird `<f>` auf L angewendet. Ist `<f>` jedoch für mehrere Argumente definiert und sollten deren Werte durch die Elemente von L dargestellt werden, dann ist stattdessen `(apply <f> <sexpr>)` zu notieren. Im Fall der Liste ist zu unterscheiden zwischen `(<f><sexpr>)` für „*single-argument functions*“ und `(apply <f> <sexpr>)` für „*multi-argument functions*“.

Da das Binden eines Symbols an eine anonyme Funktion, also die Namensvergabe für eine Funktion, eine häufige Aufgabe ist, gibt es in LISP-Systemen dazu eine syntaktische Vereinfachung. Das explizite Notieren von `lambda` entfällt bei der Kurzschreibweise. Statt

`eval<` `(define` $<$ *symbol* $>$
`(lambda` `(`$<$ *parameter*$_1$ $>$ $\ldots$ $<$ *parameter*$_n$ $>$`)` $\ldots$`))`

kann verkürzt geschrieben werden:

`eval>` `(define` `(`$<$ *symbol* $>$
$<$ *parameter*$_1$ $>$ $\ldots$ $<$ *parameter*$_n$ $>$`)` $\ldots$`)`

Bezogen auf unser Beispiel `Abbuchen` können wir daher alternativ formulieren:

```
eval> (define  Abbuchen
        (lambda
          (Alter_Saldo Abbuchungsbetrag)
          (- Alter_Saldo Abbuchungsbetrag))
      )

eval> (define
        (Abbuchen Alter_Saldo Abbuchungsbetrag)
        (- Alter_Saldo Abbuchungsbetrag))
```

Hinweis: *Syntax* `define`*-Konstrukt*
Diese syntaktische Vereinfachung des `define`-Konstruktes wird im Folgenden nicht genutzt. Die Langform dokumentiert unmissverständlich, dass die anonyme Funktion wie ein sonstiger Wert benannt wird. Sie verdeutlicht, dass im Rahmen dieser Benennung keine Unterscheidung zwischen verschiedenen Typen von symbolischen Ausdrücken gemacht wird. Ein Symbol wird stets mit der Syntax `(define` $<$ *symbol* $>$ $<$ *wert* $>$`)` gebunden. (Zur Diskussion der Vor- und Nachteile einer Unterscheidung der Bindung eines Funktionswertes von einem sonstigen Wert z. B. $\hookrightarrow$ [70] oder [3]). Zusätzlich besteht die Gefahr, dass eine LISP-Implementation die syntaktische Vereinfachung nicht in diese Langform transformiert (z. B. *Kyoto Common LISP*, $\hookrightarrow$ [195] p. 105, oder ältere *PC Scheme* Versionen, die in ein `NAMED-LAMBDA`-Konstrukt transformieren).

Wir halten fest: Das `lambda`-Konstrukt ist in zwei Rollen zu betrachten:

1. Funktionale Abstraktion, d. h. Definition einer Funktion
   (a) anonyme Funktion: `(lambda ... )`
   (b) benannte Funktion: `(define` < *symbol* > `(lambda ... ))`
2. Applikation, d. h. Anwendung einer Funktion
   (a) anonyme Funktion: `((lambda ... ) ... )`
   (b) benannte Funktion: `(` < *symbol* > `... )`

Im folgenden erweitern wir die Funktion `Abbuchen` um die Fortschreibung des Kontos. Dazu wird ein Beispielkonto `Meyer-AG` mit dem Anfangswert 1000000.00 initialisiert.

```
eval> (define Meyer-AG 1000000.00)
```

Wendet man die Funktion `Abbuchen` an, dann ist festzustellen, dass wir für die Abbuchung eine bleibende Zustandsänderung des Kontos benötigen. Die Zustandsänderung ist bisher nicht als Rechenvorschrift in dem `Abbuchen`-Konstrukt definiert.

```
eval> (Abbuchen Meyer-AG 300000.00)
        ==> 700000.0 ; erste Abbuchung

eval> (Abbuchen Meyer-AG 800000.00)
        ==> 200000.0
       ;zweite Abbuchung -
       ; schön für die Meyer-AG,
       ; aber unsinnig, denn wir erwarten
       ; den Wert -100000.00 .
```

Das `set!`-Konstrukt (engl. pronounced: „set-bang“) ändert den Wert eines Symbols.

`eval> (set!` < *symbol* >< *sexpr* >`)`
`==>` $\{<$ *value* − *sexpr* $>\}$

Wenn ein Symbol (in den erreichbaren Umgebungen) existiert, ersetzt das `set!`-Konstrukt seinen Wert durch den Wert von `<sexpr>`. Da der bisherige Wert „zerstört“ wird, bezeichnet man `set!` als ein destruktives Konstrukt. Destruktive Konstrukte verändern Zustände. Wir sind allgemein nicht an ihrem (Rückgabe-)Wert interessiert, sondern an dem Nebeneffekt, d. h. an der Zustandsänderung. Der Wert eines destruktiven Konstruktes ist daher prinzipiell nicht spezifiziert. Abhängig vom jeweiligen LISP-System wird ein Wert zurückgegeben, um dem *READ-EVAL-PRINT*-Zyklus zu genügen (↪ Abschnitt 1.1.2 S. 20).[9] Bewirkt ein Konstrukt einen Nebeneffekt, dann fügen wir der Bezeichnung dieses destruktiven Konstruktes ein „`!`“ (Ausrufezeichen) hinzu.

[9] In *PLT-Scheme* ist der Rückgabewert `#<void>`, d. h. der Wert der Applikation von `(void)`. Es gilt daher:
`(eq? (set! Foo 7) (void))` oder `(void? (set! Foo 7)) ==> #t`.

```
eval> (define Schulze-GmbH -60000.00)
eval> (set! Schulze-GmbH +1234.56)
eval> Schulze-GmbH ==> 1234.56
```

Wir können die definierte Funktion `Abbuchen`, wie folgt, benutzen, um den Kontostand bleibend zu verändern:

```
eval> Meyer-AG ==> 1000000.  ;Anfangswert
eval> (set! Meyer-AG
        (Abbuchen Meyer-AG 300000.00))
eval> Meyer-AG ==> 700000.
     ;Kontostand nach 1. Abbuchung
eval> (set! Meyer-AG
        (Abbuchen Meyer-AG 800000.00))
eval> Meyer-AG ==> -100000.
     ;Kontostand nach 2. Abbuchung
```

Diese Schreibweise ist recht umständlich. Wir ergänzen daher die Funktion zum Abbuchen direkt um die Zustandsänderung des Kontos, d. h. um den Nebeneffekt. Das Konstrukt `Abbuchen` nennen wir deshalb jetzt `Abbuchen!` (am Ende mit einem Ausrufezeichen versehen).

Um eine Funktion `Abbuchen!` verstehen zu können, erläutern wir vorab einige Aspekte von `eval` und `apply`, bezogen auf die jeweilige Umgebung. Im Abschnitt 2.5 S. 270 befassen wir uns eingehend mit der Funktion und ihrer Umgebung.

Umgebung (Namensraum)

Wenn ein Symbol mit dem `define`-Konstrukt kreiert und an einen Wert gebunden wird, dann muss das LISP-System das Symbol und den Wert speichern. Die Assoziation des Symbols zum Wert wird in einem Namensraum (engl.: *namespace*) — salopp formuliert in der Umgebung (engl.: *environment*) — aufgebaut, die zum Zeitpunkt des evaluierens des `define`-Konstruktes aktuell ist. Eine Umgebung ist konzeptionell als eine Liste vorstellbar, deren Elemente wieder Listen sind. Eine solche Liste ist eine sogenannte Assoziationsliste.

$$((<symbol_1><wert_1>)\ \ldots\ (<symbol_n><wert_n>))$$

Das Evaluieren eines `lambda`-Konstruktes führt zum Aufbau einer neuen Umgebung. Um diesen Vorgang zu verdeutlichen, definieren wir eine Funktion, deren Name keine Rolle spielt: Wir nennen sie `Foo`, so gut wie `Sinnlos`.

Hinweis: Bedeutungsloser Name
Das Symbol `foo`, oder `Foo` ist eine Silbe ohne Bedeutung, allerdings mit einer langen LISP-Historie. `Foo` wird allgemein verwendet, wenn der Name des Konstruktes unerheblich ist. Reicht ein Symbol nicht aus, dann

verwendet man häufig als weitere bedeutungslose Namen `bar` und/oder `baz`.

```
eval> (define Foo
        (lambda (x)
          (define Mein-Symbol 0)
          (set! Mein-Symbol 10)
          Mein-Symbol))
eval>  Foo ==> #<procedure:Foo>
eval> (Foo 5) ==> 10 ;Rückgabewert
                     ; von Mein-Symbol
eval> Mein-Symbol
      ==> ERROR ...
        ;Symbol ist nicht definiert in
        ; der aktuellen Umgebung
```

Weil ein `lambda`-Konstrukt eine neue Umgebung aufbaut, führt die folgende spontane Formulierung nicht zur gewünschten Zustandsänderung unseres Kontos.

```
eval> (define Abbuchen!
        (lambda
          (Konto_Name Abbuchungsbetrag)
          (set! Konto_Name ;hier wird
                  ; das Symbol benötigt
          ;eval, weil jetzt der
          ; Wert von Konto_Name
          ; benötigt wird.
          (- (eval Konto_Name)
             Abbuchungsbetrag))
           Konto_Name))
eval> Meyer-AG ==> 1000000.0
eval> (Abbuchen! 'Meyer-AG 300000.0) ==> 700000.0
eval> Meyer-AG ==> 1000000.0 ;Nicht verändert!
```

Die destruktive Veränderung des `set!`-Konstruktes betrifft die vom `lambda`-Konstrukt erzeugte Umgebung und nicht die (Ausgangs-)Umgebung in der das Symbol `Meyer-AG` den assoziierten Wert hat, der zu verändern ist. Für die gewünschte Funktion `Abbuchen!` benötigen wir ein Konstrukt zum Erzeugen einer Liste. Bisher haben wir Listen direkt notiert und nicht mit einem Konstrukt erzeugt. Jetzt nutzen wir den eingebauten Konstruktor `list`.

`list`-Konstrukt

Es hat als Wert eine Liste, gebildet aus den Werten seiner Argumente.

```
eval> (list < sexpr_1 > ... < sexpr_n >)
       ==> (< value_1 > ... < value_n >)
```

```
eval> (list 'A 'B 'C) ==> (A B C)
eval> (list) ==> ()
eval> (list (list)) ==> (())
eval> (list 'set! 'Meyer-AG 12.34)
        ==> (set! Meyer-AG 12.34)
eval> (list 'set! 'Meyer-AG (- 50 39))
       ==> (set! Meyer-AG 11)
```

Die Funktion `Abbuchen!` lässt sich mit dem `list`-Konstrukt und dem `eval`-Konstrukt, wie folgt, notieren:

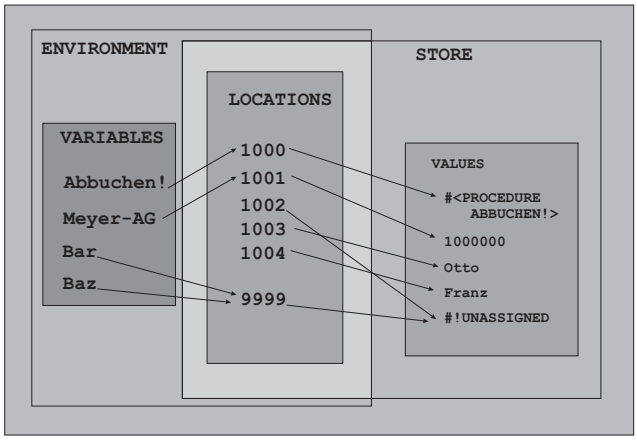
```
eval> (define Abbuchen!
        (lambda (Konto_Name Abbuchungsbetrag)
         (eval (list 'set!
                     Konto_Name
                       (- (eval Konto_Name) ;Wert
                                 ; von Konto_Name
                                Abbuchungsbetrag)))
          ))
eval> (define Meyer-AG 100000.0)
eval> (Abbuchen! 'Meyer-AG 300000.0)
eval> Meyer-AG ==> -200000.0
eval> (Abbuchen! 'Meyer-AG 400000.0)
      ==> -600000.0
```

Das `Abbuchen!`-Konstrukt, definiert auf der Basis des `lambda`-Konstruktes, gibt nicht nur einen Wert zurück, sondern verändert auch den Wert seines ersten Arguments. Die Schnittstelle (↪ Abbildung 1.11 S. 34) wird durch die beiden *lambda*-Variablen `Konto_Name` und `Abbuchungsbetrag` dargestellt. Den Nebeneffekt ruft das enthaltene `set!`-Konstrukt hervor.

Wir betrachten im Folgenden die Abbildung 1.12 S. 45, um die Symbol-Wert-Assoziation und ihre Implementation zu erläutern. Fasst man eine Symbol-Wert-Assoziation als die Benennung eines Wertes auf, dann hat das Symbol die Rolle einer Variablen. Wir bezeichnen ein Symbol als Variable, wenn wir seine Wert-Assoziation betonen wollen.

In Programmiersprachen benennen Variable häufig Werte („Objekte“) in zwei Schritten (↪ z. B. [39] p. 224). Eine Variable verweist auf einen Speicherplatz (hier in `LOCATIONS`). Dieser Speicherplatz enthält umgangssprachlich den Wert; präzise formuliert: Den Zeiger, der auf den Wert zeigt. Gewöhnlich sprechen wir verkürzt von dem Wert der Variablen. Wesentlich ist, dass es aufgrund dieses zweistufigen Benennungsverfahrens auch zwei Wege für eine Änderung gibt. Was die Variable benennt, kann wie folgt geändert werden:

1. Durch Bindung (engl.: *binding*) kann die Variable zu einem anderen Speicherplatz (hier: `LOCATIONS`) zeigen.



Legende:

`9999` ≡ Bezeichnet den Platz, der „ungebunden“ entspricht.
`#!UNASSIGNED` ≡ Bezeichnet den Wert („Objekt“), der ”„nicht initialisiert“ entspricht.

Abbildung 1.12: Zweistufiges Verfahren zur Wert-Benennung

2. Durch Zuweisung (engl.: *assignment*) kann ein anderer Wert an dem Platz gespeichert werden, an den die Variable gebunden ist.

Den Speicher (hier: `STORE`) kann man sich als ein Lexikon mit einem Schlagwort für jeden Platz (in `LOCATIONS`) vorstellen. Das Lexikon dokumentiert, welcher Wert an diesem Platz gespeichert ist. Viele Eintragungen weisen aus, dass kein Wert an ihrem Platz gespeichert ist, d. h. der Platz ist nicht initialisiert, weil er (noch) nicht benutzt wird. Eine Umgebung ist ebenfalls ein Lexikon. Es enthält Schlagwörter für jede ihrer Variablen, die ausweisen, an welchen Platz die Variable gebunden ist. Manche Eintragungen dokumentieren, dass die Variable ungebunden ist, d. h. die Variable ist an keinen eigenen Platz gebunden, sondern an den Platz, der „ungebunden“ entspricht.

Aus der zeitlichen Perspektive sind daher zwei Aspekte zu betrachten:

1. Namen, die ihre Werte („Quantitäten“) ändern und

2. Werte, die ihre Namen ändern.

In der klassischen Programmierung, z.B. in der COBOL-Welt, konzentriert man sich auf die Namen und verfolgt schrittweise ihre Werteänderung. Im Kontext der `lambda`-Bindung ist die zweite Sicht zweckmäßig. Es geht um Werte, die neu benannt werden. Die `lambda`-Variable

ist ein neuer Name für den Wert eines Arguments. So gesehen ist das `lambda`-Konstrukt ein **Umbenennungs**-Konstrukt (↪ [172]).

Exkurs: *Aliasname*.

Die Möglichkeit einem existierenden Objekt mehrere Namen geben zu können, verursacht Transparenzprobleme. Ist das Objekt über jeden Namen modifizierbar, dann müssen wir stets alle Namen verfolgen, um eine Aussage über den Objektzustand machen zu können.

```
eval> (define Foo "Ok?")
eval> (alias Bar Foo) ;nicht definiert
                      ; in Scheme!
eval> (set! Bar "Neuer Wert")
eval> Foo ==> "Neuer Wert"  ;modifiziert mit
                            ; Bar-Änderung
```

Wird in Scheme ein `lambda`-Konstrukt evaluiert, dann kann die entstandene Funktion auf die Umgebung zugreifen, in der sie evaluiert wurde. Das durch `#<procedure>` repräsentierte (Funktions-)Objekt ist verknüpft mit seiner Definitions-Umgebung. Die Umgebung und die Funktion bilden eine Einheit. Bildlich gesehen umhüllt die Umgebung die Funktion. Für die Verknüpfung von Umgebung und Funktion hat sich daher der englische Begriff *closure* (deutsch: Hülle oder auch Abschließung) durchgesetzt. Die Umgebung ist „geschlossen“ im Hinblick auf die Variablen (engl.: *closed over the variables*), weil diese nicht außerhalb bzw. ohne diese Umgebung zugreifbar sind.

In Scheme kann der Namensraum explizit angegeben und manipuliert werden. Dazu verfügt Scheme, z. B. *PLT-Scheme* (DrScheme), über vielfältige Konstrukte wie z. B. `current-namespace`, `make-base-namespace`, `make-base-empty-namespace` und `make-empty-namespace`.

Im folgenden Beispiel binden wir in einem `lambda`-Konstrukt den aktuellen Namensraum an das Symbol `env`. Dann wird die Definition der Fuktion `Baz` in diesem Namensraum definiert. Dazu nutzen wir das `eval`-Konstrukt. Da dieses vorab von seinen Argumenten die Werte ermittelt, konstruieren wir das Konstrukt `(define Baz (lambda (x y) (+ x y)))` mit Hilfe des `list`-Konstruktes (↪ S. 43).

```
eval> (define Foo
        (lambda ()
          (define env (make-base-namespace))
          (eval (list 'define
                      'Baz
                      '(lambda (x y) (+ x y)))
                env)
          env))
eval> Baz ==> ERROR ...
         ;reference to an identifier
         ; before its definition: Baz
eval> (eval '(Baz 2 4) (Foo)) ==> 6
```

Das *Closure*-Konzept wurde 1964 von *P.J. Landin* im Zusammenhang mit seinem Vorschlag für die Auswertung von *Lambda*-Ausdrücken angegeben (↪ [113]). Es ist die Lösung für Probleme beim Umgang mit Funktionen als Argumente und als Rückgabewerte von Funktionen. Diese sogenannten *Funarg*-Probleme treten auf, wenn die Funktion beim Aufruf nicht mehr über ihre Definitionsumgebung verfügt (↪ z. B. [132]).

Exkurs: *Funarg*-Probleme.
Man spricht vom *downward funarg*-Problem, wenn eine *lambda*-Variable an eine Funktion gebunden wird; d. h. eine Funktion ist Wert eines Argumentes. Das *upward funarg*-Problem entsteht, wenn der Wert einer Funktion eine Funktion ist, d. h. eine Funktion gibt eine anschließend anzuwendende Funktion zurück.

Klassische LISP-Systeme können die Funktion nur mit der Umgebung abarbeiten, die bei ihrem Aufruf besteht. Die Definitions-Umgebung ist nicht mit der Funktion verknüpft. Da die Umgebung beim Aufruf von der Situation zum jeweiligen Aufrufungszeitpunkt abhängt, ist diese stets eine andere. Es besteht eine dynamische Situation; man spricht daher von einer „*dynamischen Bindung*".

Das *Closure*-Konzept ist demgegenüber eine „*statische Bindung*", da sich die Definitionsumgebung für die Funktion nicht ändert. Im Scheme-Kontext bezeichnet man eine *statische Bindung* als *lexikalische Bindung* und die *dynamische Bindung* als *fluid binding*.

Wenn eine Funktion angewendet wird, dann sind die Werte der Argumente in den Funktionskörper zu „übernehmen". Die folgenden drei Schritte stellen ein Erklärungsmodell für diese Übernahme dar (↪ Abbildung 1.12 S. 45):

1. Schritt: Binden der *lambda*-Variablen.
   (a) Es werden neue Plätze (in `LOCATIONS`) für jede *lambda*-Variable der Funktion belegt.
   (b) Die Werte der Argumente des Funktionsaufrufes werden in den korrespondierenden neuen Plätzen gespeichert. Das führt zu einem neuen `STORE`.
   (c) Die *lambda*-Variablen werden dann an die neuen Plätze (in `LOCATIONS`) gebunden. Dies entspricht einer Neubenennung von Werten.
   (d) Diese Bindung bewirkt eine neue Umgebung (`ENVIRONMENT`). In der neuen Umgebung zeigen die *lambda*-Variablen auf die neuen Plätze (in `LOCATIONS`).
2. Schritt: Evaluieren in neuer Umgebung.
   Die Rechenvorschrift der Funktion wird in der neuen Umgebung in Verbindung mit dem neuen `STORE` evaluiert.

3. Schritt: Aktualisieren der alten Umgebung.
   Nach vollzogener Auswertung werden die geschaffenen Bindungen der *lambda*-Variablen wieder aufgelöst. Die alte Umgebung wird wieder zur aktuellen Umgebung.

Bildlich formuliert, entspringt die *lexikalische Bindung* folgender Überlegung: Wenn jemand einen Satz (Funktion) formuliert, dann unterstellt er seinen Worten (Symbolen) eine bestimmte Bedeutung. Er unterstellt z. B. die Erläuterungen im Sinne des Lexikons „Duden“. Sein Satz wird möglicherweise falsch interpretiert, wenn der Leser ein anderes Lexikon annimmt. Die Verknüpfung der Funktion mit dem Definitionslexikon soll die korrekte Interpretation sichern.

1.1.5 Zusammenfassung: `eval`, `apply` und `lambda`

LISP (*LISt Processing*) ist eine formale Sprache, eine dialogorientierte Software-Entwicklungsumgebung, eine Menge vordefinierter Konstrukte und ein Formalismus zur Spezifikation. LISP-Konstrukte haben eine einfache Syntax (Grammatik) und (als Dialekt Scheme) eine einfache Semantik (Bedeutung). Symbole, Zahlen, Zeichen, Zeichenketten und Listen sind symbolische Ausdrücke. Jeder symbolische Ausdruck ist ein syntaktisch korrekt konstruiertes LISP-„Programm“. Die Regeln zur Auswertung symbolischer Ausdrücke (EVAL-Regeln) definieren die Semantik. Sie bilden das `eval`-Konstrukt. Den *READ-EVAL-PRINT*-Zyklus notieren wir wie folgt:

`eval>` < *sexpr* > `==>` < *sexpr* >

Ist eine Liste zu evaluieren, dann bestimmt das erste Element, wie die restlichen Listenelemente behandelt werden.

Benennt z. B. das erste Element eine Funktion, dann sind die restlichen Elemente Argumente, die vor der Übernahme in den Funktionskörper evaluiert werden. Abweichend von der üblichen Notation $f(x,y)$ verschiebt LISP die Funktion f in die Klammer und verwendet als Trennzeichen das Leerzeichen statt dem Komma; also `(f x y)`.

Die Funktionsapplikation kann durch das `apply`-Konstrukt explizit angegeben werden.

Ist das erste Element ein spezielles Symbol, z. B. `define` oder `if`, dann hängt es vom jeweiligen Symbol ab, ob Argumente evaluiert werden oder nicht. Das spezielle Symbol `quote` blockiert die Auswertung. Das Symbol `eval` führt zur Anwendung der *EVAL*-Regeln. Das `eval`-Konstrukt dient als Gegenstück zum `quote`-Konstrukt.

Der Wahrheitswert „nicht wahr“ wird repräsentiert durch `#f` oder *false*. Alle Werte ungleich „nicht wahr“ werden als Wahrheitswert „wahr“ (`#t` oder `true`) aufgefasst.

Ein Konstrukt ist ein symbolischer Ausdruck, gekennzeichnet durch Schnittstelle, (Rückgabe-)Wert(e) und gegebenenfalls durch Nebeneffekt(e). Das `lambda`-Konstrukt dient zur Definition von eigenen Konstrukten. Die *lambda*-Variablen bilden die Schnittstelle; die Rechenvorschrift setzt sich aus eigenen und/oder eingebauten Konstrukten zusammen.

Der Wert eines Symbols ist von der Umgebung abhängig, in der das Symbol evaluiert wird. Jedes `lambda`-Konstrukt baut eine eigene Umgebung auf. Es übernimmt damit die Aufgabe der Neubenennung von Werten.

Charakteristische Beispiele für Abschnitt 1.1

```
eval> (define Müller-OHG 'Kunde)
eval> (define Meyer-AG (quote Müller-OHG))
eval> (eval Meyer-AG) ==> Kunde
eval> Meyer-AG ==> Müller-OHG
eval> Müller-OHG ==> Kunde

eval> (eval '(eval '(eval '(eval '(+ 1 2 3 4)))))
       ==> 10
eval> (+ 1 (+ 2 (+ 3 (+ 4)))) ==> 10


eval> (define Zustimmung? "nie und nimmer")

eval> (if Zustimmung? "Wählen!" "Nicht wählen!")
      ==> "Wählen!" ;Der Wert von
                    ; "nie und nimmer"
                    ; ist ungleich false.
                    ; Die Bedingung gilt
                    ; als erfüllt, daher
                    ; wird der "True-Fall"
                    ; evaluiert.

eval> (define Zustimmung? (+ 2 3))

eval> (if Zustimmung? "Wählen!" "Nicht wählen!")
      ==> "Wählen!" ;Der Wert 5 ist
                    ; ungleich false.
                    ; Evaluiert wird daher der
                    ; "True-Fall".

eval> (define einzahlen!
        (lambda (Konto_Name
                 Einzahlungsbetrag)
           (eval (list 'set! Konto_Name
                       (+ (eval Konto_Name)
```

```
                              Einzahlungsbetrag)))
            ))
eval> (define Meyer-AG 100000.0)
eval> (einzahlen! 'Meyer-AG 300000.0)
eval> (einzahlen! 'Meyer-AG 800000.0)
eval>  Meyer-AG ==> 1200000.0

eval> (define + -) ==> +  ;Achtung!!
             ; Es wird ein eingebautes
             ; Symbol modifiziert.
eval> (+ 3 4) ==> -1
```

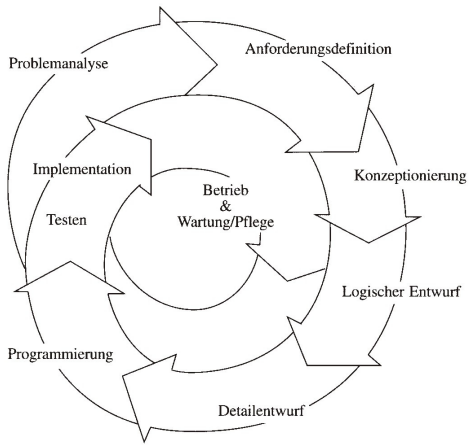
1.2 Kombinieren von Konstrukten

Eine spezifizierte Aufgabe zu programmieren bedeutet, passende Konstrukte selbst zu definieren und diese mit fremden Konstrukten zu kombinieren. Solches „Handhaben“ von Konstrukten im Sinne eines Problemlösungsvorganges ist nicht isoliert betrachtbar und bewertbar. Ob eine Lösung (Konstruktion) zweckmäßig ist oder nicht, zeigt sich nicht nur in Bezug zur spezifizierten Aufgabe, die es gilt (möglichst!) korrekt abzubilden. Zusätzlich soll die Konstruktion leicht an eine geänderte Situation anpassbar sein. Die zu erreichende Software-Qualität bezieht sich auf die Gesamtheit von Eigenschaften und Merkmalen; auf die Gebrauchstauglichkeit (funktionale Qualität), die Entwicklungsfähigkeit und die Übertragbarkeit (realisierungstechnische Qualität).

Das Einbeziehen der Anforderungen des gesamten Software-Lebenszyklusses ist zwingend erforderlich (Näheres ↪ Abschnitt 3.2 S. 403). Die Bewertung einer Konstruktion umfasst alle Phasen, z. B. : Problemanalyse, Anforderungsdefinition, Konzeptionierung, logischer Entwurf, Detailentwurf, Programmierung, Testen, Implementation, Betrieb und Wartung/Pflege/ Fortentwicklung (↪ Abbildung 1.13 S. 51). Sie wirft unterschiedlichste Fragen auf: ökonomische, gesellschaftliche, soziale, organisatorische, juristische und viele mehr.

Unsere Thematik *Konstruktion* zielt auf die software-technischen Fragen. Wir konzentrieren uns auf die Probleme, die man üblicherweise Software-Ingenieuren zur Lösung überträgt. Viele relevante Konstruktionsaspekte wie z.B. die Sozialverträglichkeit werden nicht diskutiert. Zieht man einen Vergleich zum Bau eines Fahrgastschiffes, dann werden Aspekte der Seetüchtigkeit und Sicherheit anhand der Dicke der Stahlwände geklärt, während die Frage, ob es überhaupt genügend Fahrgäste gibt, dabei nicht berücksichtigt wird.

Schon diese eingegrenzte Betrachtung des klassischen Lebenszyklusses (↪ z. B. [21]) zwingt uns permanent zur Entscheidung: Ist diese Konstruktion oder eine ihrer möglichen Alternativen zu wählen?



Legende:
Problemanalyse, Anforderungsdefinition, Konzeptionierung, logischer Entwurf, Detailentwurf, Programmierung, Testen, Implementation Betrieb & Wartung / Pflege bilden einen Zyklus.

Abbildung 1.13: Lebenszyklus-Skizze des Produktes *Software*

Beispiel: Entscheidung für eine Lösung

In einem Programm zur Verwaltung von Kunden sei zu überprüfen, ob vorab die Frage nach dem Namen und dem Alter des Kunden beantwortet wurde. Welches der beiden Programmfragmente ist zu bevorzugen und warum?

Lösung 1

```
eval> (define a #t)  ;Anwort des Namens
eval> (define b #f) ==> B ;Antwort des Alters
eval> (define foo
        (lambda (x y) (if x y #f)))
          ...
eval> (foo a b) ==> #f
```

Lösung 2

```
eval> (define Antwort-Name  true)
eval> (define Antwort-Alter false)
          ...
eval> (and Antwort-Name Antwort-Alter) ==> #f
```

Offensichtlich ist Lösung 2 mit dem `and`–Konstrukt leichter zu verstehen als Lösung 1, obwohl beide Lösungen ähnlich aufgebaut sind. Die Lösung 2 weist einprägsame (mnemotechnisch besser gewählte) Symbole auf und greift auf das `and`-Konstrukt der Booleschen Algebra zurück, das als häufig verwendetes Konstrukt schneller durchschaut wird als das hier selbstdefinierte Äquivalent, das als `foo`-Konstrukt formuliert ist.

Die Auswahlentscheidung zwischen Lösungsalternativen ist allgemein nicht trivial. Es ist schwierig festzulegen, welche Aufgabenteile (Leistungen) in einem Konstrukt zusammenzufassen sind, und wie das Konstrukt im Detail zu formulieren ist. Für das Problem, eine komplexe Aufgabe zweckmäßig in kleinere Einheiten (Konstrukte) zu modularisieren (gliedern), gibt es selten eine Lösung, die unstrittig gegenüber ihren möglichen Alternativen nur Vorteile bietet. Die Menge der konkurrierenden und zum Teil sogar sich widersprechenden Forderungen verhindert, dass es den eindeutigen „Favoriten“ gibt. Die Entscheidungen zur Bildung eines Konstruktes sind daher Kompromissentscheidungen. Eine tragfähige Kompromisslösung ist einerseits von den allgemein bewährten und anerkannten Konstruktionsregeln und andererseits von individuellen Bevorzugungen und Erfahrungen geprägt.

Der folgende Abschnitt stellt anhand des Beispiels einer Kontenverwaltung die Bildung von Konstrukten dar. Im Kapitel II wird mit diesem Beispiel die Modularisierung im Zusammenhang mit verschiedenen Abbildungsoptionen (Liste, Vektor, Zeichenkette etc.) weiter vertieft.

1.2.1 Interaktion: Operand — Operation

Bei der Bildung eines Konstruktes orientiert man sich intuitiv an Fachbegriffen des jeweiligen Aufgabengebietes. Sie bezeichnen z. B. reale Objekte (Gegenstände) oder konzeptionelle Einheiten. In unserem Beispiel einer Kontenverwaltung könnten daher erste Orientierungspunkte sein: Ein Konto, eine Buchung oder ein realer Gegenstand wie der Aktenschrank im Büro des Buchhalters. Ein Konstrukt bildet dann Eigenschaften und ihre Veränderungen einer solchen „realen“ Einheit in Wechselwirkung mit anderen „realen“ Einheiten ab. Es modelliert die Eigenschaften und Aktionen. Die gedankliche Verknüpfung zwischen „realen“ Einheiten und den Sprachkonstrukten ist die Basis für das Formulieren und Verstehen von Programmen. Ein direkter Bezug zwischen „realer“ Einheit und Konstrukt, z. B. hergestellt durch eine Namensgleichheit, erleichtert daher das Verstehen (Näheres ↪ Abschnitt 3.1.1 S. 377).

Die Matrix (↪ Tabelle 1.3 S. 53) verdeutlicht Alternativen zur Bildung von Konstrukten. Sie skizziert in der Realität vorgefundene Arbeiten (Aktivitäten) als Operatoren, bezogen auf Operanden, deren Zustand manipuliert wird. Entsprechend dieser Matrix kann man ein Konstrukt bilden, das ausgehend von einer bestimmten Operation alle möglichen Operanden zusammenfasst. In diesem Fall ist die Matrixspalte die

Oper-anden	Operationen			
	`Neuanlegen`	`Löschen`	`Anmahnen`	...
`Kunde`	x_{11}	x_{12}	x_{13}	...
`Lieferant`	x_{21}	x_{22}	x_{23}	...
`Kassenkonto`	x_{31}	x_{32}	x_{33}	...
⋮	⋮	⋮	⋮	⋮

Legende:
Zeilen- oder spaltenorientierte Bildung von Konstrukten.

Tabelle 1.3: Matrix: Operanden — Operationen

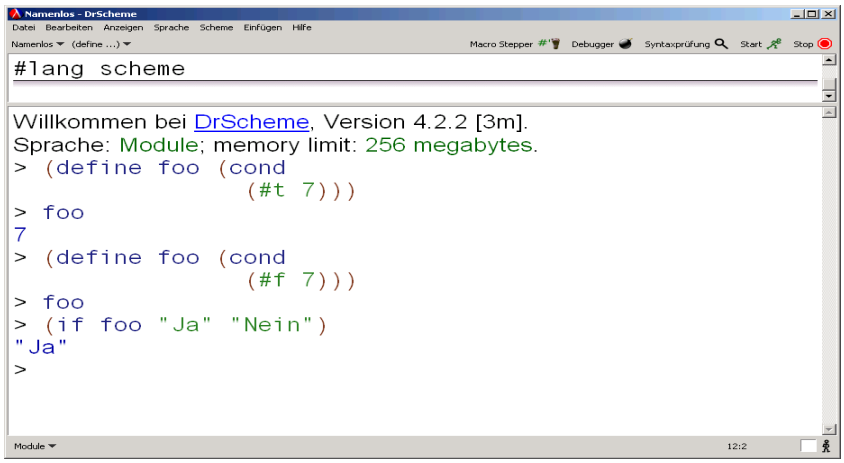
Klammer für die Zusammenfassung zu einem Konstrukt. Zu definieren sind dann die Konstrukte: `Neuanlegen`, `Löschen`, `Anmahnen` etc. . Das Konstrukt `Neuanlegen` ist auf `Kunde`, `Lieferant`, `Kassenkonto` etc. anzuwenden. Ist der Operand ein `Kunde`, dann ist die Operation x_{11} auszuführen. Beim Operand vom Typ `Lieferant` ist es die Operation x_{21}. Es ist daher zwischen den verschiedenen Typen von Operanden zu unterscheiden. Für diese Unterscheidung gibt es in LISP das `cond`-Konstrukt (*conditional expression*). Diese Fallunterscheidung ersetzt eine tiefe Schachtelung von `if`-Konstrukten. Bevor wir die Struktur von `Neuanlegen` definieren, wird vorab das `cond`-Konstrukt erläutert.

Konditional-Konstrukt `cond`

Das `cond`-Konstrukt wählt die zutreffende Bedingung aus einer Menge beliebig vieler Bedingungen aus. Es entspricht einer Eintreffer-Entscheidungstabelle (↪ S. 80) und hat folgende Struktur:

```
eval> (cond   (<p1>   <sexpr11>   ...   <sexpr1m>)
              (<p2>   <sexpr21>   ...   <sexpr2k>)
                      ⋮
              (<pi>   <sexpri1>   ...   <sexprij>)
                      ⋮
              (<pn>   <sexprn1>   ...   <sexprnl>)
          )   ==>     <value>
```

Eine Liste `(<`p_i`> <`$sexpr_{i1}$`> ... <`$sexpr_{ij}$`>)`, genannt Regel oder Klausel (engl.: *clause*), bildet einen Fall ab. Dabei ist `<`p_i`>` die Bedingung (das Prädikat) für den einzelnen Fall und die optionalen `<`$sexpr_{i1}$`>` bis `<`$sexpr_{in}$`>` sind die symbolischen Ausdrücke, die der Reihe nach ausgewertet werden, wenn die Bedingung `<`p_i`>` erfüllt ist. `<`p_i`>` ist erfüllt,



Legende:
Dr. Scheme ↪`http://www.plt-scheme.org/` (Zugriff: 24-Oct-2009)

Abbildung 1.14: Beispiel `cond`-Konstrukt

wenn der Wert von <p_i> ungleich `false` ist.

Der Wert des `cond`-Konstruktes, also <`value`>, wird dann nach Evaluierung von <$sexpr_{i1}$> bis <$sexpr_{ij}$> durch den zuletzt ermittelten Wert bestimmt. Falls $j > 0$, also ein <$sexpr_{ij}$> evaluiert wurde, ist <*value*> der Wert von <$sexpr_{ij}$>; ist $j = 0$, dann ist <*value*> gleich dem Wert von <p_i>. Die Bedingungen werden der Reihe nach von <p_1> beginnend geprüft, bis erstmals der Wert von <p_i> ungleich `false` ist, d. h. als wahr interpretiert wird. Die restlichen Bedingungen werden nicht ausgewertet. Ist keine Bedingung ungleich `false`, dann ist das `cond`-Konstrukt prinzipiell undefiniert. Das jeweilige LISP-System repräsentiert dann diesen „Wert“; z. B. bei *PLT-Scheme* (*DrScheme*) mit einem nicht darstellbaren Leerzeichen (↪ Abbildung 1.14 S. 54).

Beispiele zum `cond`-Konstrukt

```
eval> (define Absolutwert
        (lambda (x)
          (cond ((< x 0) (* -1 x))
                ("sonst" x))))

eval> (Absolutwert -5) ==> 5
eval> (Absolutwert  5) ==> 5

eval> (define *Weg* 'unbenutzt)
```

```
eval> ;;Anonyme Funktion mit Nebeneffekt
      ((lambda (x y)  ;Anwendung der
                      ; anonymen Funktion
        (cond ((= x y)
                 (set! *Weg* "Klausel =")
                   (* x y))
              ((< x y)
                 (set! *Weg* "Klausel x < y")
                   (+ x y))
              ((< y x)
                 (set! *Weg* "Klausel x > y")
                   (- y y))
              ( #t (set! *Weg* "Klausel sonst")
                     "Unbekannte Relation!")))
            12 23) ;Argumente
                    ==> 35 ;Rückgabewert
eval> *Weg*
       ==> "Klausel x < y" ;Bewirkter Nebeneffekt


eval> (define Otto false)
eval> ;;Keine Klausel trifft zu!
      (cond
            (Otto (list))          ; 1. Klausel
            ((< 2 1) (list))       ; 2. Klausel
            ((eval #f) (list))     ; 3. Klausel
            ) ==>
```

Beim letzten Beispiel ist der Wert des `cond`-Konstruktes unbestimmt, weil keine der drei Klauseln zutrifft. Dieser undefinierte Fall des `cond`-Konstruktes wird durch keinen Ausgabewert repräsentiert. Um ein solches „Durchlaufen“ zu vermeiden, ist es zweckmäßig, am Schluss eines `cond`-Konstruktes eine `true`-Klausel (`#t`) zu formulieren.

Hinweis: `true`-*Schlussklausel*
Jeder symbolische Ausdruck, dessen Wert ungleich `false` ist, kann als Prädikat verwendet werden, um eine Klausel zu definieren, die auf jeden Fall zutrifft, also die Aufgabe einer Sonst-Regel (*otherwise case*) übernimmt. Traditionell verwendete man das Symbol `T`. In Scheme hat der Wahrheitswert `true` die Repräsentation `#t`.

Für die Unterscheidung z. B. zwischen einem Kunden und einem Lieferanten entsprechend Tabelle 1.3 S. 53, benötigen wir passende Prädikate, damit das `cond`-Konstrukt die Auswahl der jeweiligen Operationen übernehmen kann. Für die Typerkennung unterstellen wir hier, dass die

```
eval> (define Meyer-AG
        '(Kunde "Daten zur Meyer AG"))
eval> (define Müller-OHG
        '(Kunde "Daten zur Müller OHG"))
eval> (define Schulze-GmbH
        '(Lieferant "Daten zur Schulze-GmbH"))
eval> (define Verwahrung-1
        '(Kassenkonto
           "Daten zum Verwahrkonto-1"))
```

Legende:
Beispiele für `Kunde`, `Lieferant` und `Kassenkonto` entsprechend Tabelle 1.3 S. 53.

Tabelle 1.4: Programmfragment: Kunde, Lieferant und Kassenkonto

Daten eines Kunden in Form einer Liste abgebildet sind, deren erstes Element das Symbol `Kunde` ist. Für einen Lieferanten, ein Kassenkonto etc. gibt es entsprechende Listen. Vorab definieren wir mit Programmfragment 1.4 S. 56 zwei Kunden, einen Lieferanten und ein Kassenkonto.

Um in diesem Beispiel das Symbol `Meyer-AG` als Kunde zu erkennen, benötigen wir einen Selektor für das erste Element einer Liste und ein Konstrukt, das auf Gleichheit testet. Im Zusammenhang mit dem benötigten Selektor erläutern wir zusätzlich den korrespondierenden Konstruktor.

Selektoren `car` und `cdr`

Das klassische „*Pure LISP*“ basiert auf den beiden Selektoren `car` (historisch bedingte Abkürzung für *Contents of Address part of Register*; phonetisch `[ka:]`) und `cdr` (*Contents of Decrement part of Register*; phonetisch `[kd:]`). Sie ermöglichen es, die Teile einer Liste zu selektieren. Dabei hat das `car`-Konstrukt das erste Element der Liste, d. h. den sogenannten Kopf, und das `cdr`-Konstrukt die Restliste als Wert.

Diese `car-cdr`-Sprechweise bezieht sich auf die klassische Repräsentation von Listen, d. h. auf Adressen im Rechner (↪ Abschnitt 2.2 S. 160). Sie sollte jedoch nicht den Blick dafür verstellen, dass Listen abstrakte Formalobjekte sind, deren Implementation zunächst untergeordnete Bedeutung hat. Plakativ formuliert: Bei der Konstruktion beschäftigen wir uns mit abstrakten Formalobjekten und nicht mit bekannten Adressen!

```
eval> (car Meyer-AG) ==> Kunde
eval> (car (cdr Meyer-AG))
         ==> "Daten zur Meyer AG"
eval> (car Schulze-GmbH)
        ==> Lieferant
eval> (car (cdr Schulze-GmbH))
        ==> "Daten zur Schulze GMBH"
eval> (car Verwahrung-1)
        ==> Kassenkonto
eval> (car (cdr Verwahrung-1))
        ==> "Daten zum Verwahrkonto-1"
```

Legende:
Selektion des Operanden-Typs und der zugehörenden Daten für Programmfragment 1.4 S. 56.

Tabelle 1.5: Programmfragment: Selektion des Operanden-Typs und der zugehörenden Daten

Beispiele zum `car`- und `cdr`-Konstrukt

```
eval> (cdr 'Leuphana)
       ==> ERROR ...
           ;keine Liste - quote !!!
eval> (car "Leuphana")
       ==> ERROR ...
           ;keine Liste - string !!!
eval> (car (cdr '(eins zwei drei vier)))
       ==> zwei
eval> (car (cdr (cdr (cdr
        '(eins zwei drei vier)))))
      ==> vier
```

Zur Reduzierung des Schreibaufwandes einer Kette geschachtelter Selektoren ermöglichen LISP-Systeme eine verkürzte Schreibweise in der Form `c...r`, z. B. `cadr` oder `cdddr`. Üblicherweise können `ca`*x*`r` oder `cd`*x*`r` formuliert, wobei *x* eine Folge von bis zu drei Buchstaben `a` bzw. `d` sein kann. Zusätzlich existiert ein Selektor, der auf eine angegebene Elementeposition zugreift (↪ `list-ref`-Konstrukt S. 126).

```
eval> (cadr '(eins zwei drei vier)) ==> zwei
eval> (caar '((eins) zwei))) ==> eins

eval> (ddddddr '(1 2 3 4 5 6 7 8 9))
      ==> ERROR ...
          ;ddddddr ist nicht definiert
```

```
        ; und wäre auch unzweckmäßig
```

Werden die Selektoren `car` und `cdr` auf die leere Liste angewendet, dann ist ihr Rückgabewert nicht definiert. Ihr Wert ist undefiniert, so wie es der Wert bei der Division einer Zahl durch 0 ist. Abhängig von der jeweiligen LISP-Implementation wird jedoch `null` bzw. `NIL` oder eine Fehlermeldung oder ein sonstiges Zeichen zurückgegeben. In *PLT-Scheme (DrScheme)* haben die Selektoren dann folgende Werte:

```
eval> (car ()) ==> ERROR ...
eval> (cdr ()) ==> ERROR ...
eval> (car '()) ==> ERROR ...
eval> (cdr '()) ==> ERROR ...
eval> (car ''()) ==> quote
eval> (cdr ''()) ==> (())
```

Konstruktor `cons`

Während die beiden Selektoren `car` und `cdr` eine Liste in zwei Teile zerlegen, verbindet der Konstruktor `cons` (*to construct*) zwei symbolische Ausdrücke. Das `cons`-Konstrukt hat eine Schnittstelle für zwei Argumente. Es hat als Wert eine neue Liste, wenn das zweite Argument eine Liste ist. Der Wert ist die neue Liste mit dem Wert des ersten Arguments als erstes Element dieser Liste. Kurz formuliert: `cons` fügt einen symbolischen Ausdruck als Kopf an den Anfang einer Liste. (Häufige engl. LISP-Phrase: "*I'll cons something up*").

Beispiele zum `cons`**-Konstrukt**

```
eval> (cons 'Leuphana '(Universität Lüneburg))
      ==> (Leuphana Universität Lüneburg)
eval> (cons 1  '(2  3)) ==> (1 2 3)
eval> (cons 1 (list)) ==> (1)
```

Ist der Wert des zweiten Arguments keine Liste, dann hat das `cons`-Konstrukt ein sogenanntes Punkt-Paar (engl.: *dotted pair*) als Wert, wobei der linke Teil der Wert des ersten Argumentes und dessen rechter Teil der Wert des zweiten Argumentes von `cons` ist. Die Bezeichnung "*dotted pair*" leitet sich davon ab, dass zwischen die evaluierten Argumente ein Punkt eingefügt wird.

```
eval> (cons 'noch 'nicht) ==> (noch . nicht)
```

Die externe Darstellung eines Punkt-Paares, die als Eingabe und Ausgabe verwendet wird, hat die Reihenfolge: öffnende Klammer, symbolischer Ausdruck, Zwischenraum, Punkt, Zwischenraum, symbolischer Ausdruck und dann schließende Klammer. Die Zwischenräume sind erforderlich, um ein Punkt-Paar von einer Gleitkommazahl zu unterscheiden, bzw. um den Punkt nicht als Namensteil eines Symbols aufzufassen.

```
      ;Gleitkommazahlen
eval> (+ 1.2 3.4) ==> 4.6

      ;Punkt-Paar!
eval> (cons 1.2 3.4)
       ==> (1.2 . 3.4)

      ;Punkt als Namensteil
eval> (cons 'Bremer-Lagerhaus.AG #f)
      ==> (Bremer-Lagerhaus.AG . #f)
```

Das Punkt-Paar verweist auf die interne Abbildung eines symbolischen Ausdruckes im Speicher des LISP-Systems. Bei der `cons`-Anwendung wird (in den klassischen LISP-Implementationen) eine sogenannte LISP-Zelle (auch `cons`-Zelle genannt) im Speicher mit den Adressen der Werte der beiden Argumente generiert. Eine Liste ist damit ein Spezialfall einer Punkt-Paare-Struktur (↪ Abbildung 1.15 S. 60). Sie ist entweder:

- ein `null`-Zeiger (bzw. `nil`-Zeiger) oder
- ein Punkt-Paar, dessen `car`-Teil ein symbolischer Ausdruck ist und dessen `cdr`-Teil eine Liste ist.

Für unsere Thematik „Konstruktion“ ist die Zusicherung entscheidend: Was ein Konstruktor zusammenfügt, ist ohne Verlust durch korrespondierende Selektoren zerlegbar, d.h. wiedergewinnbar.

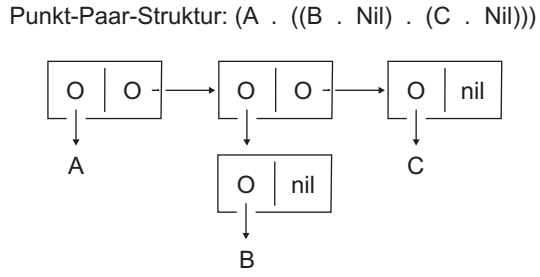
```
eval> (car (cons 'Bauteil-1 'Bauteil-2))
        ==> Bauteil-1
eval> (cdr (cons 'Bauteil-1 'Bauteil-2))
       ==> Bauteil-2
```

Die Definition von eigenen Konstruktoren und passenden Selektoren vertiefen wir im Abschnitt 2.1 S. 141.

Prädikate: `eq?`-Konstrukt und `equal?`-Konstrukt

Wenn zwei symbolische Ausdrücke miteinander verglichen werden, dann sind drei Fälle zu unterscheiden:

1. *Identität*:
   Beide Angaben bezeichnen dasselbe „Objekt“.
2. *Gleichheit*:
   Beide Angaben haben den gleichen Wert. Es sind aber prinzipiell zwei Abbildungen der „Objekte“ (Werte) feststellbar, d. h. die Werte sind an zwei verschiedenen Stellen gespeichert (Merkformel: „Zwillinge“).



Legende:

car-Part	cdr-Part	≡ cons-Zelle

o---> ≡ Zeiger (Adresse)

nil ≡ nil-Zeiger; entspricht o--->nil

Abbildung 1.15: `cons`-Zellen-Darstellung der Liste `(A (B) C)`

3. *Ungleichheit*.

Die Gleichheit stellen wir häufig im Hinblick auf eine Austauschbarkeit fest. Zweck ist es, *Gleiches durch Gleiches* zu ersetzen. Beispielsweise wollen wir den symbolischen Ausdruck `(+ 1 2)` durch die Zahl `3` ersetzen.

Sprechen wir von Gleichheit im Sinne von Nichtunterscheidbarkeit, dann haben alle Tests für beide „Objekte“ stets identische Resultate. Eine Identitätsfeststellung bei zwei Argumenten bedingt das Prüfen zweier Namen, die dasselbe „Objekt“ benennen.

Das Vergleichsprädikat im Sinne einer solchen Identitätsfeststellung heißt `eq` (abgekürzt von *equal*; phonetisch [i:kju:]) oder ähnlich z. B. `eq?` in Scheme. Bei Symbolen nimmt das `eq?`-Konstrukt Identität an, wenn beide Argumente sich auf dasselbe interne „Objekt“ im LISP-System beziehen (Prüfung der Gleichheit von Adressen ↪ Abschnitt 2.2 S. 160).

Bei Zahlatomen ist man an ihrer numerischen Wertgleichheit interessiert und nicht daran, ob das LISP-System die beiden Argumente (Zahlen) als dasselbe „Objekt“ interpretiert oder nicht. Gelöst wird dieses Problem entweder durch ein eigenständiges Vergleichskonstrukt für Zahlatome (z. B. das Symbol = bei Scheme) oder durch entsprechende Erweiterung von `eq` (↪ z. B. TLC-LISP). In modernen LISP-Systemen ist dieses Vergleichen häufig als eine sogenannte generische Funktion realisiert. Eine solche Funktion stellt selbst fest von welchem Typ die übergebenen Werte sind und aktiviert dann das zugeordnete Vergleichskonstrukt.

Sind die symbolischen Ausdrücke im Sinne der Ersetzbarkeit *Gleiches durch Gleiches* zu prüfen, dann ist das `equal?`-Konstrukt anzuwenden. Hat schon das `eq?`-Konstrukt für zwei Argumente den Wert `true`, dann

```
eval> ;;Symbol Kunde ist in
      ;; Scheme nur einmal gespeichert
      (eq? (car Meyer-AG) 'Kunde) ==> #t

eval> ;;Die Symbole sind identisch,
      ;; die Information, dass sie
      ;; Elemente einer Liste sind,
      ;; ist zweimal abgebildet,
      ;; daher keine Identität.
      (eq? Meyer-AG
        '(Kunde "Daten zur Meyer AG"))
          ==> false

eval> ;;Gleichheit von Listenstrukturen
      ;; und Listenelementen
      (equal? Meyer-AG
        '(Kunde "Daten zur Meyer AG"))
          ==> true

eval> ;;Ein Symbol ist nicht
      ;; gleich einer Zeichenkette
      (equal? 'Kunde "Kunde") ==> false
```

Legende:
↪ Tabelle 1.4 S. 56

Tabelle 1.6: Programmfragment: Erkennen des Operandentyps Kunde

hat auch das `equal?`-Konstrukt für diese Argumente den Wert `true`. (Achtung: Die Umkehrung gilt nicht!).

Mit dem `eq?`-Konstrukt sind wir in der Lage, Prädikate für die Erkennung von Kunden, Lieferanten, Kassenkonten etc. zu definieren. Da diese Prädikate zum Unterscheiden zwischen verschiedenen „Objekten“ dienen, bezeichnet man sie auch als *Diskriminatoren*. In Scheme haben Prädikate üblicherweise als Kennzeichen am Ende ihres Namens ein Fragezeichen, daher nennen wir die Prädikate `Kunde?`, `Lieferant?`, `Kassenkonto?` etc. Entsprechend der Matrix-Spalte (↪ Tabelle 1.3 S. 53) kann `Neuanlegen` mit der Struktur von Programmfragment in Tabelle 1.7 S. 62 definiert werden.

Im Programmfragment der Tabelle 1.7 S. 62 sind die Operationen `x11`, `x21` und `x31` nicht ausformuliert, sondern nur als Konstanten angegeben. Zur Ausgabe einer Fehlermeldung nutzen wir das `print`-Konstrukt. Es evaluiert sein Argument. Um zwei erläuternde Zeichenketten

```
eval> (define Kunde?
        (lambda (x)
          (eq? (car x) 'Kunde)))

eval> (define Lieferant?
        (lambda (x)
          (eq? (car x) 'Lieferant)))

eval> (define Kassenkonto?
        (lambda (x)
          (eq? (car x) 'Kassenkonto)))

eval> (define Neuanlegen
      (lambda (operand)
        (cond((Kunde? operand)        'x11)
             ((Lieferant? operand)   'x21)
             ((Kassenkonto? operand) 'x31)
             (#t (print (list
                   "Operationen für:"
                   operand
                   "noch nicht definiert!"
             )))
        )))
eval> (Neuanlegen Schulze-GmbH) ==> x21
eval> (Neuanlegen Verwahrung-1) ==> x31
eval> (Neuanlegen '(a b c)) ==>
("Operationen für:" (a b c) "noch nicht definiert!")
```

Legende:
↪ Tabelle 1.4 S. 56

Tabelle 1.7: Programmfragment: Operation `Neuanlegen` als Konstruktbasis

(Strings) und den Wert von `operand` auszugeben, nutzen wir das `list`-Konstrukt.

Hinweis: `newline`-Konstrukt.
Für einen Zeilenvorschub in der Ausgabe (Konsole) gibt es das `newline`-Konstrukt. Damit lässt sich leicht das in Programmiersprachen übliche `println`-Konstrukt wie folgt selbst definieren:

```
eval> (define println
        (lambda (x) (print x) (newline)))
eval> (println "1. Zeile") (println "2. Zeile")
"1. Zeile"
"2. Zeile"
eval>
```

Bei einem komplexen Problem führt die Zusammenfassung von Operationen entsprechend der Matrix-Spalten (↪ Tabelle 1.3 S. 53) zu einer Hierarchie von Konstrukten. Diese Hierarchie ist geprägt durch die Operationen (Aktionen) des jeweiligen Aufgabengebietes. Beispielsweise käme man zur Operation `Neuanlegen` über ein hierarchisch höheres Konstrukt. Wir nehmen an, dass diese Auswahloperation den Namen `Verwalten` habe. Das Konstrukt `Verwalten` ermittelt dann anhand seines Operanden, dass `Neuanlegen` auszuführen sei.

So entsteht eine operations-geprägte Konstruktion. Sie ist gegliedert (modularisiert) in Operationen. Im Mittelpunkt der systema(naly)tischen Überlegungen stehen die Aktionen (Prozesse, Vorgänge).

Wären nach Fertigstellung der Konstruktion weitere Operanden, wie z. B. Sachkonten, einzubeziehen, dann sind in den Operationen `Neuanlegen`, `Löschen` etc. jeweils die entsprechenden Funktionen für die Sachkonten einzubauen. Dies ist im Falle vieler bestehender Operationen ein sehr aufwendiges und fehleranfälliges Unterfangen, weil fertiggestellte Konstrukte z. B. `Neuanlegen` modifiziert werden müssen. Damit sind neue Fehler für schon programmierte Operandentypen nicht auszuschließen.

Sind solche Änderungen wahrscheinlich, dann empfiehlt es sich, die Konstrukte aus der Matrixzeilen-Sicht zu entwickeln, statt sie aus der Matrixspalten-Sicht zu bilden. Exemplarisch ist das Konstrukt `Kunde` als Programmfragment in Tabelle 1.8 S. 64 definiert. Es skizziert die Operanden-Sicht in stark vereinfachter Art.

Auch hier führen die Abstraktionsschritte zu einer Hierarchie von Konstrukten. Beim Ergänzen um den Operanden `Sachkonto` werden die bestehenden Konstrukte wie `Kunde`, `Lieferant` etc. nicht berührt. Das Zusammenfassen aller Operationen bietet daher ein Modularisierungskonzept, das ein Hinzufügen von neuen Konstrukten im Sinne von Operanden erleichtert. Ist allerdings eine neue Operation z. B. `Umbenennen` später zu ergänzen, dann sind die Konstrukte `Kunde`, `Lieferant`, `Kassenkonto` etc. zu modifizieren.

```
eval> (define Kunde
        (lambda (operation)
          (cond((Neuanlegen? operation) 'x11)
                ((Löschen?   operation) 'x12)
                ((Anmahnen?  operation) 'x13)
                (#t (print (list
                  "Für Kunden ist:"
                  operation
                  "noch nicht definiert!")))
          )))

eval> (define Neuanlegen?
        (lambda (x)(eq? (car x) 'Neuanlegen)))

eval> (define Meyer-AG
         (Kunde '(Neuanlegen
           "Daten zur Meyer AG")))
```

Legende:
↪ Tabelle 1.4 S. 56

Tabelle 1.8: Programmfragment: Operand `Kunde` als Konstrukt-Basis

So entsteht eine operanden-geprägte Konstruktion. Sie hat einen objekt-orientierten Aufbau. Im Mittelpunkt der systema(naly)tischen Überlegungen stehen die Operanden im Sinne von Objekten (Näheres dazu ↪ Abschnitt 2.8 S. 327).

Die Frage, ob die Lösung *Operation als Konstrukt-Basis* (↪ Tabelle 1.7 S. 62) oder die Lösung *Operand als Konstrukt-Basis* (↪ Tabelle 1.8 S. 64) zweckmäßiger ist, bedingt eine Prognose über die Fortentwicklung des Softwareproduktes. Das skizzierte Beispiel Kontenverwaltung verdeutlicht, dass unsere Konstruktions-Entscheidungen im Kontext des gesamten Software- Lebenszyklus zu treffen sind. Die Erfüllung einer funktionalen Anforderung z. B. das korrekte Abbilden einer Buchung ist notwendig, aber nicht hinreichend. Hinzukommen muss die Abdeckung von erwartbaren Anforderungen aus den späteren Phasen des Lebenszyklus.

In unserem Kontext „Konstruktion“ ist stets zu unterscheiden, ob es um die Realisierung relativ kleiner Konstrukte (Menge einzelner „Anweisungen/Befehle“) oder um große Konstrukte (ganze Programmeinheiten) geht. Im ersten Fall ist die Reihenfolge der Aktivierung einzelner Anweisungen/Befehle zu diskutieren. Im zweiten Fall geht es um die Koordination von ganzen Programmteilen (Moduln), also um abgeschlossene Folgen von Anweisungen/Befehlen. Zunächst widmen wir uns der Realisierung kleiner Konstrukte, die häufig als „Programmierung im Kleinen“ (engl.: *programming-in-the-small*) bezeichnet wird (↪ [48]) Wir fangen also klein an und skizzieren mit wachsendem LISP-Fundus Verknüpfungstechniken für das „Programmieren im Großen“ (engl.: *programming-in-the-large*).

1.2.2 Kontrollstrukturen

Legen wir fest, wann, unter welchen Bedingungen und wie oft einzelne Konstrukte auszuwerten sind, so definieren wir die Kontrollstruktur (engl.: *control structure*). Sie beschreibt, ob und in welcher Reihenfolge Programm(teil)e ausgeführt werden. Man bezeichnet sie korrekterweise auch als *Steuerungsstruktur* oder auch als *Ablaufsteuerung*. In der Terminologie imperativ geprägter Programmiersprachen (z. B. COBOL oder Pascal, Näheres ↪ Abschnitt 2.1 S. 141) spricht man vom Steuerungsfluss oder Kontrollfluss (engl.: *flow of control*).

Kombinierbar sind Konstrukte durch eine Hintereinanderausführung. Diese Verkettung nennt man im Zusammenhang mit Funktionen *Komposition*. In der Mathematik wird die Komposition zweier Funktionen f und g notiert als „$f\ g$“ und gelesen als „f verkettet mit g“ oder „g eingesetzt in f“ oder „f nach g“. Diese Komposition ist in LISP als `(f (g x))` zu formulieren mit f und g als Funktionen, die für ein Argument definiert sind (engl.: *unary functions*). Der Wert dieser Komposition ist der Wert von `(f y)`, wobei y der Wert von `(g x)` ist. Die Komposition `(f (g x))`

ist nur dann sinnvoll, d. h. nicht undefiniert, wenn x die Schnittstelle von g und y die Schnittstelle von f erfüllen. Anders formuliert: Die Komposition bedingt, dass x in der Definionsmenge von g und die Wertemenge von g in der Definitionsmenge von f enthalten sind.

Beispiele für die Komposition

```
eval> (define add2 (lambda (zahl) (+ 2 zahl)))
eval> (define mul3 (lambda (zahl) (* 3 zahl)))
eval> (add2 (mul3 1)) ==> 5
eval> (mul3 (add2 1)) ==> 9
eval> (define a2m3
        (lambda (zahl)
          (add2 (mul3 zahl))))
eval> (a2m3 1) ==> 5
eval> ;;Komposition zweier anonymer Funktionen
      ((lambda (zahl) (+ 2 zahl))
         ((lambda (zahl) (* 3 zahl)) 1)) ==> 5
```

Elementare Kontrollstrukturen sind:

1. die Folge (Sequenz)
2. die Auswahl (Selektion) und
3. die Wiederholung (Iteration).

Bevor wir ihre Realisierung diskutieren, sind nochmals die bisher erläuterten Konstruktionsmittel in Backus-Naur-Form (↪ folgenden Exkurs) als Tabelle 1.9 S. 67 und 1.10 S. 68 zusammengestellt. Sie zeigt atomare Komponenten (wie Symbol und Zahl), Grundprozeduren (wie `cons`, `car` und `cdr`) und zusätzlich elementare Kontrollstrukturen. Z. B. ermöglicht das `lambda`-Konstrukt das Verknüpfen von Konstrukten als Sequenz.

Exkurs: *Backus-Naur-Form*

Die von *John W. Backus* und *Peter Naur* definierte Notation ist geeignet, die Syntax einer Programmiersprache zu spezifizieren. Eine *BNF*-Metasprachvariable, auch syntaktische Einheit genannt, ist eine Variable, deren Wert auf Zeichen zurückführbar ist, die in der gegebenen Sprache (hier LISP) zulässig sind. Aus Klarheitsgründen sind die *BNF*-Metasprachvariablen in spitze Klammern $< \ldots >$ eingeschlossen.

Damit unterscheiden sich diese *nonterminal symbols* (kurz *non terminals*) von den Symbolen in LISP, die auch als *terminal symbols* (kurz *terminals*) bezeichnet werden.

Die *BNF*-Metasprachgleichheit, hier dargestellt durch $\equiv$, kann aufgefasst werden als „ist konstruierbar aus“. Dabei wird eine Wahlmöglichkeit im

BNF-Variable		Spezifikation
$< sexpr >$	$\equiv$	$< atom > \mid < liste >$
$< liste >$	$\equiv$	$(\{< sexpr >\})$
$< atom >$	$\equiv$	$< symbol > \mid < zahlatom > \ldots$
		;und weitere Typen z. B.
		; Zeichen oder Zeichenkette
$< symbol >$	$\equiv$	$< buchstabe > \mid < sonderzeichen >$
		$\mid < symbol >< buchstabe >$
		$\mid < symbol >< sonderzeichen >$
		$\mid < symbol >< ziffer >$
		$\mid < ziffer >< symbol >$
$< zahlatom >$	$\equiv$	$< numeral >$
		$\mid + < numeral > \mid - < numeral >$
$< numeral >$	$\equiv$	$< ziffer > \{< ziffer >\}$
		$\mid \ldots$;und weitere Typen
		; z. B. 12.34
$< buchstabe >$	$\equiv$	A | B | C | ... | Z
		| a | b | c | ... | z
$< sonderzeichen >$	$\equiv$	* | < | > | ! | ...
		;und weitere Typen
$< ziffer >$	$\equiv$	0 | 1 | 2 | ... | 9

Legende:
Backus-Naur-Form ↪ Exkurs Abschnitt 1.2.2 S. 66
Zeichen hinter einem Semikolon in einer Zeile sind Kommentar.
↪ Tabelle 1.10 S. 68.

Tabelle 1.9: Erörterte Konstruktions-Mittel (Teil 1)

Sinne einer Oder-Verknüpfung durch den senkrechten Strich | dokumentiert. Eckige Klammern [...] kennzeichnen die Möglichkeit, das Eingeklammerte weglassen zu können. Geschweifte Klammern {...} dokumentieren, dass das Eingeklammerte beliebig oft wiederholt oder aber auch ganz weggelassen werden kann.

Die Tabellen 1.9 S. 67 und 1.10 S. 68 definieren auszugsweise ein LISP-System; es umfasst keinesfalls alle Konstrukte eines LISP-Dialektes wie Scheme. Wir können jedoch mit ihnen wesentliche Charakteristika diskutieren. So kann z. B. kein Konto mit dem Namen 1234 entsprechend unserem Konto `Meyer-AG` (↪ S. 26) gebildet werden.

```
eval> (define 1234
        (Kunde
          '(Neuanlegen
             "Daten zu Konto 1234")))
      ==> ERROR ...
         ;Fehlerhaftes Symbol für define
```

BFN-Variable $<$ *sexpr* $>\equiv$	
(quote $<$ *sexpr* $>$)	;Kreieren einer Konstanten
| (cons $<$ *sexpr* $><$ *liste* $>$)	;Einfügen in eine Liste
| (car $<$ *liste* $>$)	;Selektion des ersten
	; Elementes
| (cdr $<$ *liste* $>$)	;Selektion der Restliste
| (list { $<$ *sexpr* $>$ })	;Bilden einer Liste
| (define $<$ *symbol* $><$ *sexpr* $>$)	;Binden eines Symbols
| (set! $<$ *symbol* $><$ *sexpr* $>$)	;Modifizieren einer Bindung
| (cond { ($<$ *sexpr* $>$	;Fallunterscheidung
{ $<$ *sexpr* $>$ }) })	
| (lambda ({ $<$ *symbol* $>$ })	;Definition einer
$<$ *sexpr* $>$ { *sexpr* })	; Funktion
| (apply $<$ *sexpr* $><$ *liste* $>$)	;Applikation einer Funktion
| (eval $<$ *sexpr* $>$)	;Auswertung in einer
	; Umgebung
| ($<$ *funktion* $>$	;Komposition
($<$ *funktion* $>$ { $<$ *sexpr* $>$ }))	

Legende:
Backus-Naur-Form ↪ Exkurs Abschnitt 1.2.2 S. 66
Zeichen hinter einem Semikolon in einer Zeile sind Kommentar.
↪ Tabelle 1.9 S. 67.

Tabelle 1.10: Erörterte Konstruktions-Mittel (Teil 2)

Die Zahlatome sind von Symbolen (auch Literalatome genannt) zu unterscheiden. Nur mit Symbolen ist das Kreieren einer Wert-Assoziation in einer Umgebung möglich. Zahlatome haben sich selbst als Wert; stellen aber keine Variable dar.

Eine Funktion kann daher nicht mit einem Zahlatom benannt werden. Auch die Schnittstelle der eigenen Konstrukte, gebildet durch die `lambda`-Variablen, erfordert Symbole. Zahlatome sind dort nicht verwendbar. Wir stellen fest: Für Zahlatome gibt es Restriktionen im Vergleich zu Literalatomen (Symbolen). Zahlatomen fehlt die sogenannte „*first-class*"-Eigenschaft (↪ Exkurs Abschnitt 1.2.2 S. 69). So kann die arithmetische Funktion, die den Wert ihres Argumentes um 1 erhöht, nicht `+1` heißen, weil `+1` als Zahlatom interpretiert wird. Die Umkehrung dieser Zeichenfolge ist in Scheme ein Symbol. Die Funktion ist daher mit `1+` benennbar.

```
eval> (1+ 2) ==> 3          ;1+ ist ein Symbol
eval> (+1 2) ==> ERROR ... ;+1 ist kein Symbol
eval> (+ 1 2) ==> 3         ;+ ist ein Symbol
```

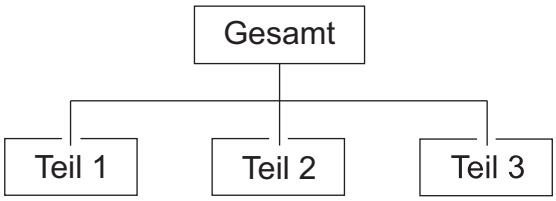
Exkurs: Begriff *First-class*-Objekt
Im LISP-Kontext haben Objekte (Werte) dann *first-class*-Eigenschaften (auch *first-class-citizen* genannt), wenn folgende Bedingungen erfüllt sind (z. B. ↪ [39], p. 224):

1. Objekte haben eine Identität. Diese ist unabhängig von jedem Namen, unter dem Objekte gekannt werden können. So kann ein Objekt auch mehrere Namen haben. (Diese Eigenschaft fehlt Zahlatomen.)
2. Objekte können anonym bleiben. Z. B. evaluiert das `lambda`-Konstrukt zu einer anonymen Funktion.
3. Objekte können ohne Verlust ihrer Identität überall gespeichert werden; als Werte von Variablen oder in komplexen Datenstrukturen.
4. Objekte können sowohl Argument als auch Rückgabewert einer Funktion sein.
5. Objekte „sterben" nicht. Ein Objekt wird unbenutzbar (unzugreifbar) und damit zwecklos. Der von ihm belegte Speicherplatz kann vom LISP-System als wieder verfügbar erklärt werden, vorausgesetzt, die Unbenutzbarkeit wird erkannt. Das LISP-System stellt fest, ob ein Objekt als „Müll" (engl.: *garbage*) zu betrachten ist. Algorithmen, die dies erkennen, bezeichnet man als *garbage-collection*-Algorithmen.

R. Popplestone spricht in diesem Kontext von „*items*", die fundamentale Rechte besitzen (Näheres ↪ [2]).

Sequenz

Wir nehmen an, ein symbolischer Ausdruck `Gesamt` setzt sich aus den Teilen `Teil-1`, `Teil-2` und `Teil-3` zusammen, wobei diese Teile eben-



Legende:
M. A. Jackson-Notation (↪ [98])

Abbildung 1.16: Grafische Darstellung einer Sequenz

```
eval> (define Gesamt (lambda ()
                       Teil-1
                       Teil-2
                       Teil-3))
eval> (define Teil-1 "<sexpr> für Modul Teil-1")
eval> (define Teil-2 "<sexpr> für Modul Teil-2")
eval> (define Teil-3 "<sexpr> für Modul Teil-3")
eval> (Gesamt) ==> "<sexpr> für Modul Teil-3"
        ;Rückgabewert ist der Wert von Teil-3
```

Tabelle 1.11: Programmfragment: Sequenz von Symbolen

falls symbolische Ausdrücke sind. Vorstellbar ist, dass `Gesamt` ein Programm ist, das aus den Moduln `Teil-1`, `Teil-2` und `Teil-3` besteht. Aufgrund der Abbildungsgleichheit von Programm und Daten ist auch annehmbar, dass `Gesamt` eine Datenstruktur abbildet, z. B. eine Anschrift, die aus der Folge Straße (`Teil-1`), Postleitzahl (`Teil-2`) und Wohnort (`Teil-3`) besteht. Eine Sequenz mit dem Namen `Gesamt` und den Teilen `Teil-1`, `Teil-2`, `Teil-3` stellt Abbildung 1.16 S. 70 grafisch dar.

Diese Darstellungsform wurde von *M. A. Jackson* insbesondere für den Programmentwurf vorgeschlagen (↪ [98]). Zur Erläuterung der Selektion (↪ S. 80) und der Iteration (↪ S. 89) wird ebenfalls die Jackson-Notation genutzt. Die abgebildete Sequenz sei die Zerlegung (Modularisierung) eines Programms. Das sogenannte Laufenlassen des Programms `Gesamt` bedeutet das Abarbeiten der Moduln `Teil-1`, `Teil-2` und `Teil-3`. Wir formulieren daher eine Funktion `Gesamt` aus den Teilen `Teil-1`, `Teil-2` und `Teil-3` (↪ Programmfragment in Tabelle 1.11 S. 70):

```
;;;Gesamt und die Moduln Teil-1, Teil-2 und Teil-3
;;; sind als Funktionen definiert.
eval> (define Gesamt (lambda ()
                       (Teil-1)
                       (Teil-2)
                       (Teil-3)))
eval> (define Teil-1 (lambda ()
                       "<sexpr> für Modul Teil-1"))
eval> (define Teil-2 (lambda ()
                       "<sexpr> für Modul Teil-2"))
eval> (define Teil-3 (lambda ()
                       "<sexpr> für Modul Teil-3"))
```

Tabelle 1.12: Programmfragment: Sequenz von Funktionsanwendungen

Im Programmfragment der Tabelle 1.11 S. 70 werden die Werte von `Teil-1`, `Teil-2` und `Teil-3` zum Zeitpunkt ihrer Definition ermittelt. Stände an der Stelle von `"<sexpr> für Modul Teil-3"` z. B. der symbolische Ausdruck `(sin (+ 0.2 0.3))`, dann wird dieses Sinusberechnungsprogramm beim Evaluieren des `define`-Konstruktes ausgeführt, wobei `Teil-3` an den Rückgabewert des `sin`-Konstruktes gebunden wird, hier `0.479425538604203`. Die Anwendung von `Gesamt` führt nur zum Ermitteln der Bindung von `Teil-3`. Dieser wahrscheinlich ungewollte Effekt ist vermeidbar, indem wir an den Modul einen „gequoteten“ symbolischen Ausdruck (eine Konstante) binden; z. B. für `Teil-3`:

```
eval> (define Teil-3 '(sin (+ 0.2 0.3)))
```

Verfahren wir für `Teil-1` und `Teil-2` entsprechend, dann ist die Sequenz `Gesamt` wie folgt zu definieren:

```
eval> (define Gesamt
        (lambda ()
          (eval Teil-1)
          (eval Teil-2)
          (eval Teil-3)))
```

Statt das `quote`-Konstrukt einzusetzen, können wir die Moduln `Teil-1`, `Teil-2` und `Teil-3` auch als `lambda`-Konstrukte definieren, so dass erst die Applikation von `Gesamt` die Auswertung der Moduln bewirken soll.

Im Programmfragment in Tabelle 1.12 S. 71 werden die Moduln `Teil--1`, `Teil-2` und `Teil-3` nur abgearbeitet. Ihnen werden vom (Haupt-)-

Modul `Gesamt` keine Werte übergeben. Um Werte über eine Schnittstelle von `Gesamt` für die sequentielle Abarbeitung der Moduln `Teil-1`, `Teil-2` und `Teil-3` bereitzustellen, definieren wir Gesamt mit `lambda`-Variablen, z. B. mit den Parametern `P_X` und `P_Y`.

```
eval> (define Gesamt
        (lambda (P_X P_Y)
          (Teil-1)
          (Teil-2)
          (Teil-3)))
eval> (Gesamt Argument-1 Argument-2)
        ==> "<sexpr> für Modul Teil-3"
```

Wir können die Abarbeitungsfolge `Teil-1`, `Teil-2` und `Teil-3` auch so interpretieren, dass ihr Ergebnis den Namen `Gesamt` hat. Der Wert von Gesamt repräsentiert die vollzogene Sequenz.

```
eval> (define Gesamt
        ((lambda (P_X P_Y)
           (Teil-1) (Teil-2) (Teil-3))
            Argument-1 Argument-2))
```

Jetzt wird die anonyme Funktion auf den Wert der beiden Argumente `Argument-1` und `Argument-2` angewendet, wobei mögliche Nebeneffekte z. B. aufgrund von `print`-Konstrukten sofort zum Tragen kämen. Der Wert der Sequenz wird durch das `define`-Konstrukt mit dem Symbol `Gesamt` assoziiert. Auch hier können wir durch `quote`- und `eval`-Konstrukte den Zeitpunkt der Ausführung festlegen (↪ Abschnitt 1.2.3 S. 103).

```
eval> (define Gesamt
        '((lambda (P_X P_Y)
            (Teil-1) (Teil-2) (Teil-3))
              Argument-1 Argument-2))
eval> (eval Gesamt)
      ==> "<sexpr> für Modul Teil-3"
```

Anonyme Funktion Bei der Anwendung einer anonymen Funktion, insbesondere wenn diese durch ein umfangreiches `lambda`-Konstrukt mit vielen `lambda`-Variablen dargestellt wird, ist die Zuordnung der Argumente zu den `lambda`-Variablen unübersichtlich. In LISP gibt es deshalb eine syntaktische Verbesserung in Form des `let`-Konstruktes.

```
eval> (let ((< variable_1 > < argument_1 >)
                    ⋮
            (< variable_n > < argument_n >))
            < body >)    ==> < value >
```

ist zurückführbar auf:

```
eval> ((lambda (< variable_1 >
                    ⋮
                < variable_n >)
            < body >)
        < argument_1 > ... < argument_n >)
             ==> < value >
```

Hinweis: `let`-Konstrukt
Das `let`-Konstrukt ist häufig als Makro-Konstrukt realisiert. Makros werden in zwei Schritten abgearbeitet. Zunächst wird aus einem Makro ein neuer symbolischer Ausdruck konstruiert. Diesen Schritt nennt man *expandieren*. Im zweiten Schritt wird der entstandene Ausdruck evaluiert. In *PC Scheme* ist das Ergebnis des ersten Schrittes mittels `EXPAND-MACRO` darstellbar (Näheres ↪ Abschnitt 2.5.3 S. 293). Damit können wir zeigen zu welchem Ausdruck das `let`-Konstrukt expandiert.

```
PC-Scheme-eval> (EXPAND-MACRO
 '(let ((A 1) (B 2)) (Foo A B)))
    ==> ((lambda (A B) (Foo A B)) 1 2)
```

`let`-Konstrukt

Das `let`-Konstrukt entspricht der Applikation eines `lambda`-Konstruktes. Wir können es als ein Konstrukt betrachten, das zunächst lokale Variablen definiert und mit diesen Bindungen seinen Funktionskörper $< body >$ auswertet.

```
eval> (let ( ; Binden der 1. lokalen Variablen
             (< lokaleVariable_1 >< sexpr_1 >)
                    ⋮
            ; Binden der n-ten lokalen Variablen
            (< lokaleVariable_n >< sexpr_n >) )
           < body >)
                 ==> < value >
```

Den syntaktischen Vorteil des „`let`-Schreibstils“ gegenüber dem „`lambda`-Schreibstil“ verdeutlicht das folgende Beispiel:

```
eval> ((lambda (W X Y Z)
         (list (car (cdr W))
               (car (cdr X))
               (car (cdr Y))
               (car (cdr Z))))
               '(Means M) '(of O)
               '(Requirements R)
               '(Engineering E))
```

```
        ==> (M O R E)
eval> (let ((W '(Means M))
            (X '(of O))
            (Y '(Requirements R))
            (Z '(Engineering E)))
           (list (car (cdr W))
                 (car (cdr X))
                 (car (cdr Y))
                 (car (cdr Z))))
        ==> (M O R E)
```

Intuitiv vermutet man in der Bindung der lokalen Variablen eines `let`-Konstruktes die Möglichkeit, eine Sequenz zu konstruieren. Da das `let`-Konstrukt einer Applikation eines `lambda`-Konstruktes entspricht, ist der Vorgang der Auswertung der Argumente vom Vorgang der Bindung der `lambda`-Variablen an die Werte zu unterscheiden. Bezogen auf das `let`-Konstrukt formuliert: Es werden zunächst alle Werte der lokalen Variablen in einer prinzipiell beliebigen Reihenfolge ermittelt, dann findet die Bindung der jeweiligen Variablen statt.

Da die Reihenfolge dieser Wertermittlung nicht allgemeingültig festliegt (sondern sogar als Ansatz für eine Parallelverarbeitung in LISP dient), sollten wir beim `let`-Konstrukt keine Sequenz unterstellen, wie im folgenden Beispiel:

```
eval> (let (
        (Foo (if (print "1. Phase")
                 "Foo-Value" #f))
        (Bar (if (print "2. Phase")
                 "Bar-Value" #f)))
        (newline)
        (list Foo Bar)) ==>
"1. Phase""2. Phase"
("Foo-Value" "Bar-Value")
```

Zu berücksichtigen ist, dass die Ermittlung aller Werte vor der Bindung an die `lambda`-Variablen erfolgt. Beim `let`-Konstrukt sind sie daher bei der Wertermittlung nicht zugreifbar.

```
eval> (let ((x 3)
            (y (* 2 x)))
           (+ x y))
        ==> ERROR ...
         ;Das Symbol x ist nicht definiert.
```

Hilfsweise können wir die Sequenz durch Schachtelung von `let`-Konstrukten abbilden, die nur eine lokale Variable ausweisen (↪ *Curryfizieren* einer Funktion, Abschnitt 2.5.1 S. 277):

```
eval> (let ((x 3))
           (let ((y (* 2 x)))
```

```
                (+ x y)))
        ==> 9
```

In LISP-Systemen gibt es eine übersichtlichere Lösung durch das `let*`-Konstrukt. Dieses bindet die lokalen Variablen der Reihe nach. Zunächst wird der Wert für die erste Variable ermittelt und dann wird diese gebunden. Danach wird jede neue Bindung in der Umgebung ausgewertet, die durch die vorhergehenden lokalen Variablen erweitert wurde.

```
eval> (let* ((x 3)
             (y (* 2 x)))
            (+ x y))
        ==> 9
```

Hinweis: Implementation des `let*`-Konstruktes
Das `let*`-Konstrukt ist häufig wie das `let`-Konstrukt als Makro-Konstrukt implementiert (Näheres ↪ Abschnitt 2.5.3 S. 293). Es expandiert unser obiges Beispiel, wie folgt:

```
PC-Scheme-eval> (EXPAND-MACRO
                  '(let* ((x 3) (y (* 2 x)))
                         (+ x y)))
          ==> ((lambda (x)
                 (let* ((y (* 2 x))) (+ x y)))
               3)
```

Das `let*`-Konstrukt ermöglicht die Abbildung der Sequenz `Teil-1`, `Teil-2` und `Teil-3` (↪ Abbildung 1.16 S. 70), wie folgt:

```
eval> (define Gesamt
        (let* ((Teil-1 "<sexpr> für Teil-1")
               (Teil-2 "<sexpr> für Teil-2")
               (Teil-3 "<sexpr> für Teil-3"))
              Teil-3 ))
```

Entsprechend unserer Diskussion über die funktionale Alternative ist die Sequenz auch wie folgt definierbar:

```
eval> (define Gesamt
        (let* (
          (Teil-1 (lambda () "<sexpr> für Teil-1"))
          (Teil-2 (lambda () "<sexpr> für Teil-2"))
          (Teil-3 (lambda () "<sexpr> für Teil-3")))
          (Teil-1) (Teil-2) (Teil-3)))
```

Solange bei der Ermittlung des Wertes für eine lokale Variable kein Bezug auf die Variable selbst genommen wird, ist die Reihenfolge *erst Wertermittlung (Auswertung des Argumentes), dann Bindung* ausreichend. Soll jedoch ein rekursiver Bezug möglich sein, dann muss die Variable bei ihrer Wertermittlung in der Umgebung der Auswertung schon existieren. Notwendig sind dann folgende Schritte:

1. Erweiterung der Auswertungsumgebung um die Variable (dabei kann die Variable an einen beliebigen Wert gebunden sein),
2. Ermittlung des Wertes für diese Variable und
3. Bindung der Variablen an den Wert in derselben Umgebung.

Diese Schrittfolge simuliert das folgende `let`-Konstrukt:

```
eval> (let ((Foo '*UNDEFINIERT*)) ;Schritt 1
           (set! Foo  ;Schritte 2 und 3
             (lambda ( ... ) ... foo ...))
           (Foo ...) ... ) ==> ...
```

In vielen LISP-Dialekten, so auch in Scheme, heißt das äquivalente Konstrukt `letrec` , wobei der Postfix `rec` die Abkürzung für *rec*ursiv ist. Wir behandeln das `letrec`-Konstrukt bei der Erörterung der Rekursion (↪ Abschnitt 1.3 S. 113). Mit dem Fixpunktoperator `Y` skizzieren wir eine Implementationsalternative (↪ Abschnitt 2.5.1 S. 277).

Das `letrec`-Konstrukt deckt die Sequenz ebenso ab wie das `let*`-Konstrukt. Zu beachten ist dabei die Reihenfolge für die Bindung einer Variablen. Im ersten Schritt wird die Variable in der Umgebung vermerkt und überdeckt damit eine globale Defintion, obwohl ihr „Startwert" im Prinzip nicht zugreifbar sein sollte.[10] Folgende Formulierungen bauen auf diesem Startwert auf. Wir vermeiden derartige Konstruktionen, zumal sie darüber hinaus leicht zu Fehlern führen.

```
eval> (define Foo 'Bar)
eval> (letrec ((Foo (list Foo Foo)))
               Foo)
      ==> (#<undefined> #<undefined>)

eval> (letrec ((Foo (* 2 Foo)))
        (print "Nicht OK!"))
      ==> ERROR ...;Nichtnumerischer Operand für
                   ; arithmetische Funktion
```

Die bisherige Diskussion der Sequenz fußt auf dem `lambda`-Konstrukt. Gezeigt wurde, dass die Sequenz als Applikation der anonymen Funktion formulierbar ist:

```
eval> ((lambda (P_X P_Y)
               Teil-1 Teil-2 Teil-3)
         Argument-1 Argument-2)
      ==> ...
```

Ein LISP-System verfügt für die Sequenz über ein „spezielles" Konstrukt (im Sinne von `eval`-Regel 3, ↪ S. 23). In Scheme ist es das `begin`-Konstrukt. Ist das Symbol `begin` das erste Listenelement, dann

[10] Z. B. ist in *PC Scheme* dieser Startwert jedoch gleich `NIL`.

werden die Argumente der Reihe nach ausgewertet. Der Wert des letzten Argumentes ist der Wert des gesamten `begin`-Konstruktes.

`eval>` `(begin` $< sexpr_1 > \ldots < sexpr_n >$`)`
`==>` $< valuesexpr_n >$
entspricht damit:

```
eval> ((lambda ()
```
$< sexpr_1 > \ldots < sexpr_n >$`))`
`==>` $< valuesexpr_n >$

Wir können uns daher das `lambda`-Konstrukt als ein Konstrukt vorstellen, das ein implizites `begin`-Konstrukt enthält. Eine implizit formulierte Sequenz ist in einigen LISP-Konstrukten implementiert, z. B. im `cond`-Konstrukt (— nicht jedoch im `if`-Konstrukt, ↪ Programmfragment in Tabelle 1.13 S. 79 —).

Die Sequenz ist auch aufgrund der festen Reihenfolge, die `eval`-Regel 2 vorgibt, abbildbar (↪ S. 22). Danach sind erst die Argumente auszuwerten, ehe die Funktion darauf angewendet wird. Wir definieren daher ein `lambda`-Konstrukt mit einem formalen Parameter `*DUMMY*`, so dass die Angabe eines Argumentes möglich ist. Die Applikation dieser anonymen Funktion bildet dann die Sequenz dadurch ab, dass $< sexpr_1 >$ das Argument bildet und $< sexpr_2 >$ der Funktionskörper ist.

```
eval> ((lambda (*DUMMY*)
```
$< sexpr_2 >$`)` $< sexpr_1 >$`)`
`==>` $< valuesexpr_2 >$

Sicherzustellen ist allerdings, dass die Bindung von `*DUMMY*` an $< valuesexpr_1 >$ keinen Einfluss auf die Berechnung von $< valuesexpr_2 >$ hat. Anders formuliert: Die `lambda`-Variable `*DUMMY*` darf nicht als freie Variable im Ausdruck $< sexpr_2 >$ vorkommen.

Durch die Komposition entsprechender `lambda`-Konstrukte ist eine Sequenz mit n-Elementen abbildbar. Eine Sequenz mit drei Elementen ist dann wie folgt zu notieren:

```
eval> ((lambda (*DUMMY*)
          ((lambda (*DUMMY*)
```
$< sexpr_3 >$`)` $< sexpr_2 >$`))` $< sexpr_1 >$`)`
`==>` $< valuesexpr_3 >$

```
eval> ((lambda (*DUMMY*)
        ((lambda (*DUMMY*) 3) 2)) 1)
          ==> 3
```

Im Zusammenhang mit dem `begin`-Konstrukt ist das `begin0`-Konstrukt zu nennen. Es wertet seine Argumente der Reihe nach aus und gibt als Rückgabewert den Wert des ersten Argumentes zurück. Da die Zählung der Argumente intern üblicherweise mit Null beginnt, wurde an den Namen `begin` einfach eine Null gehängt.

`eval>` `(begin0` $< sexpr_1 > \ldots < sexpr_n >$`)`
`==>` $< valuesexpr_1 >$

Das `begin0`-Konstrukt lässt sich als syntaktische Vereinfachung von folgendem `let`-Konstrukt auffassen:

`eval>` `(begin0` $< sexpr_0 > \ldots < sexp_n >$`)`
`==>` $< valuesexpr_1 >$
entspricht:
`eval>(let ((x` $< sexpr_0 >$`)`
`(y (lambda ()` $< sexpr_1 > \ldots < sexpr_n >$`)))`
`(begin (y) x))`
`==>` $< valuesexpr_1 >$

Hinweis: Implementation des `begin0`-Konstruktes
Das `begin0`-Konstrukt ist häufig als Makro-Konstrukt realisiert. Der `begin0`-Makro führt zu einem `lambda`-Konstrukt mit einem vom LISP-System generierten Symbol als `lambda`-Variable.

```
PC-Scheme-eval> (EXPAND-MACRO
                  '(begin0 'Teil-1
                           'Teil-2
                           'Teil-3))
                ==>  ((lambda (%00000)
                        (begin
                          (quote Teil-2)
                          (quote Teil-3)
                       %00000))
                       (quote Teil-1))
```

Beispiel zum `begin0`-Konstrukt

Definiert ist ein Konstrukt zum Stapeln von Werten (Objekten) nach dem Prinzip „*Last In, First Out*“ (kurz: LIFO-Prinzip oder auch *Stack*-Prinzip genannt). Die Zugriffsfunktion `pop!` gibt das oberste Stack-Element, während die Funktion `push!` das Objekt „kellert“, d.h. in den Stack einspeichert.

```
eval> (define *STACK* '(Mueller-OHG Schulze-GmbH))
eval> (define push! (lambda (object)
                      (set! *STACK*
```

```
;;;;Sequenz-Beispiel mit explizit zu
;;;; notierendem begin-Konstrukt
eval> (define *Kunden-Anzahl* 0)
eval> (define *Lieferanten-Anzahl* 0)
eval> (define Schulze-GmbH
        '(Lieferant "Daten zur Schulze-GmbH"))                    .
eval> (define Kunde? (lambda (x)
                       (eq? (car x) 'Kunde)))
eval> (define Lieferant?
        (lambda (x) (eq? (car x) 'Lieferant)))                    .
;;;;Fall 1: Eine Selektion mit Sequenzen, die ein
;;;; explizites begin-Konstrukt erfordern.
eval> (define Zählung
        (lambda (operand)
          (if (Kunde? operand)
           (begin (set! *Kunden-Anzahl*
                        (+ 1 *Kunden-Anzahl*))
                  (print "Bearbeite Kunde!"))
           (if (Lieferant? operand)
            (begin (set! *Lieferanten-Anzahl*
                         (+ 1 *Lieferanten-Anzahl*))
                   (print "Bearbeite Lieferant!"))
            (print "Weder Kunde noch Lieferant!")))))
;;;;Fall 2: Nutzung der impliziten Sequenz im cond
eval> (define Zählung (lambda (operand)
        (cond ((Kunde? operand)
                (set! *Kunden-Anzahl*
                  (+ 1 *Kunden-Anzahl*))
                (print "Bearbeite Kunde!"))
              ((Lieferant? operand)
                (set! *Lieferanten-Anzahl*
                  (+ 1 *Lieferanten-Anzahl*))
                (print "Bearbeite Lieferant!"))
              (#t (print
                    "Weder Kunde noch Lieferant!")))))
eval> (Zählung Schulze-GmbH)
        ==> "Bearbeite Lieferant!"
```

Tabelle 1.13: Programmfragment: Sequenz mit `begin`-Konstrukt

```
                        (cons object *STACK*))
                       "Durchgeführt!"))
eval> (define pop! (lambda ()
                     (begin0 (car *STACK*)
                       (set! *STACK*
                         (cdr *STACK*)))))
eval> (push! 'Meyer-AG) ==> "Durchgeführt!"
eval> (pop!) ==> Meyer-AG
eval> (pop!) ==> Mueller-OHG
```

Wird beim `begin`-Konstrukt eine Sequenz von symbolischen Ausdrücken notiert, die keine Nebeneffekte bewirken, dann ist nur die Auswertung des letzten symbolischen Ausdruckes notwendig. Die Auswertung aller vorhergehenden Ausdrücke ist einsparbar. Bei effizienten LISP-Implementationen wird diese Optimierungsmöglichkeit genutzt.

```
eval> (define *STACK*
        '(Mueller-OHG Schulze-GmbH Otto-KG))
eval> (begin (cdr *STACK*)
             (cdr (cdr *STACK*))
             (cdr (cdr (cdr *STACK*))))
        ==> ()
```

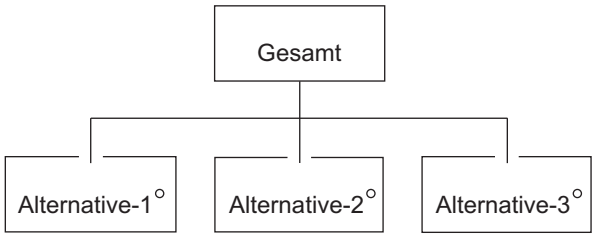
Dieses `begin`-Beispiel lässt sich auf den folgenden Ausdruck optimieren:

```
eval> (cdr (cdr (cdr *STACK*)))
         ==> ()
```

Selektion

Umfasst ein Modul zwei oder mehr Teile, von denen im Anwendungsfall nur ein Teil zutrifft, dann ist eine Selektion (Auswahl) abzubilden. Die Abbildung 1.17 S. 81 zeigt für den symbolischen Ausdruck `Gesamt` eine Selektion in *Jackson-Notation* (↪ [98]). Sein Wert ergibt sich entweder aus dem Modul `Alternative-1` oder `Alternative-2` oder `Alternative-3`. Der Kreis „°“ in der rechten oberen Ecke unterscheidet die Selektion von der Sequenz. Für die Abbildung bietet sich das `cond`-Konstrukt an. Liegt nur eine Selektion zwischen zwei Strukturen vor, ist das `if`-Konstrukt geeignet (↪ Programmfragment in Tabelle 1.13 S. 79).

Anhand der Entscheidungstabelle (engl.: *decision table*), die die Tabelle 1.14 S. 82 darstellt, erörtern wir die Selektion. Eine Entscheidungstabelle (kurz: ET) besteht aus einem Bedingungsteil und einem Aktionsteil. Die Bedingungen (im Beispiel B1, B2, B3 und B4) werden für den Selektionsvorgang der Reihe nach auf ihren Wahrheitswert hin geprüft. Die Regeln (im Beispiel R1, R2, ..., R7) verknüpfen den Bedingungsteil mit dem Aktionsteil. Sie dokumentieren, bei welchen Wahrheitswerten welche Aktionen (im Beispiel A1, A2, ..., A5) durchzuführen



Legende:
M. A. Jackson-Notation (↪ [98])

Abbildung 1.17: Grafische Darstellung einer Selektion

sind. Trifft stets nur eine Regel nach Prüfung der Bedingungen zu, dann handelt es sich um eine Eintreffer-ET, andernfalls um eine Mehrtreffer-ET (↪ z. B. Abbildung 3.7 S. 397). Beschränken sich die Wahrheitswerte auf „Ja“ (Abkürzung „J“), „Nein“ (Abkürzung „N“) und „Irrelevant“ (Abkürzung „–“), dann handelt es sich um eine begrenzte ET; begrenzt, weil die Definition der Bedingungen sich nicht in den Regelanzeigerteil hineinzieht. Eine `ET Verkehrsampel` mit Zuständen wie `ROT`, `GELB` und `GRÜN` als Angaben in den Regeln wäre nicht begrenzt.

Für eine begrenzte ET wird folgende Notation verwendet (gemäß DIN 66241 ↪ [51]):

- im Entscheidungsteil:

  J ≡ Die Bedigung ist erfüllt.

  N ≡ Die Bedingung ist nicht erfüllt.

  – ≡ Die Anwendung der Regel ist nicht davon abhängig, ob die Bedingung erfüllt oder nicht erfüllt ist. Man kann dieses Zeichen als Dokumentation der Irrelevanz betrachten. Die Bedingung kann geprüft werden, nur ist das Prüfungsergebnis unerheblich.

  # ≡ Ausschlussanzeiger, d. h. für die Entscheidung ist die Bedingung nicht definiert; ihr Wahrheitswert kann nicht ermittelt werden. (Ist hier wegen der Vollständigkeit angegeben und wird im Beispiel nicht verwendet.)

- im Aktionsteil:

  X ≡ Die Aktion ist durchzuführen.

  blank ≡ Die Aktion ist nicht durchzuführen.

Bei einer begrenzten Eintreffer-Entscheidungstabelle ist feststellbar, ob für alle möglichen Bedingungskombinationen auch Regeln genannt

ET-Auszahlung-buchen		R1	R2	R3	R4	R5	R6	R7
B1	Kontonummer angegeben?	J	J	J	J	N	N	N
B2	Kontonummer $\leq$ 40000	J	J	J	N	–	–	–
B3	Betrag $\geq$ 50.00 €	–	J	N	–	J	N	N
B4	Zweck genannt?	J	N	N	–	–	J	N
A1	Dem Chef vorlegen		X					X
A2	Zurück an die Fachabteilung schicken					X		
A3	Umkontieren auf Konto „Sonstiges“			X			X	
A4	Vorgang an Abteilung abgeben				X			
A5	Buchung vollziehen	X		X			X	

Legende:
Formale Vollständigkeitsüberprüfung:

2 einwertige Regeln (R2 und R3)	$\hookrightarrow$ 2 Fälle
3 zweiwertige Regeln (R1, R6 und R7)	$\hookrightarrow$ 6 Fälle
2 vierwertige Regeln (R4 und R5)	$\hookrightarrow$ 8 Fälle
	$\sum$ 16 Fälle
	$\equiv$ 16 Regeln

Tabelle 1.14: Begrenzte Eintreffer-Entscheidungstabelle

sind (formale Vollständigkeit). Bei einer solchen ET mit 4 Bedingungen sind 16 Regeln notwendig, wenn nur die Wahrheitswerte „J“ und „N“ verwendet werden. Enthält die ET Regeln mit Irrelevanz-Zeichen („–“), dann können diese mehrwertigen Regeln leicht auf mehrere einwertige zurückgeführt werden (↪ Tabelle 1.14 S. 82).

Die ET-Auszahlung-buchen (↪ Tabelle 1.14 S. 82) definiert, welche Aktionen in einem aktuellen Fall durchzuführen sind. Um die Daten eines aktuellen Falles zu erhalten, wird eine einfache Dialogschnittstelle konstruiert. Dazu verwenden wir das `read`-Konstrukt und das `print`-Konstrukt.

Bei einer Antwort „N“ für die Bedingung B1 ist es nicht mehr entscheidungsrelevant, den aktuellen Wert der Bedingung B2 zu erfragen. Es sollten daher nicht die aktuellen Werte aller Bedingungen der Reihe nach abgefragt werden. Ist eine Bedingung nicht relevant für den weiteren Auswahlprozess zur Bestimmung der zutreffenden Regel, dann ist sie zu übergehen. Anders formuliert: Um ungewollte Nebeneffekte (hier: Abfragen) zu vermeiden, dürfen irrelevante Bedingungen nicht evaluiert werden.

Zwei Anwendungsbeispiele für die in Tabelle 1.15 S. 83 definierte ET-Auszahlung-buchen:

```
eval> (ET-Auszahlung-buchen) ==>
"Ist die Kontonummer angegeben? (J/N)"J
"Ist die Kontonummer kleiner gleich 40000 ? (J/N)"N
```

```
;;;;ET-Auszahlung-buchen
;;;; Bedingungen
eval> (define B1 (lambda () (print
         "Ist die Kontonummer angegeben? (J/N)")
        (read))) ==> B1
eval> (define B2 (lambda () (print
          "Ist die Kontonummer kleiner gleich 40000 ? (J/N)")
        (read)))
eval> (define B3 (lambda () (print
          "Ist der Betrag groesser gleich 50.00 EUR ? (J/N)")
        (read)))
eval> (define B4 (lambda () (print
          "Ist der Zweck angegeben ? (J/N)")
        (read)))
;;;;Aktionen
eval> (define A1 (lambda () (print
          "Dem Chef vorlegen")))
eval> (define A2 (lambda () (print
          "Zurueck an die Fachabteilung schicken")))
eval> (define A3 (lambda () (print
          "Umkontieren auf Konto SONSTIGES")))
eval> (define A4 (lambda () (print
          "Vorgang an Abteilung 2 abgeben")))
eval> (define A5 (lambda () (print
          "Buchung vollziehen")))
;;;;Regeln
eval> (define  ET-Auszahlung-buchen (lambda ()
        (cond((eq? (B1) 'J)
               (cond((eq? (B2) 'J)
                      (cond((eq? (B4) 'J) (A5))
                            (#t (cond((eq? (B3) 'J) (A1))
                                      (#t (A3)(A5))))))
                     (#t (A4))))
              (#t (cond((eq? (B3) 'J) (A2))
                        (#t (cond((eq? (B4) 'J)
                                  (A3) (A5))
                                 (#t (A1)))))))))
```

Legende:
↪ Tabelle 1.14 S. 82. Bedingungen und Aktionen außerhalb des Konstruktes `ET-Auszahlung-buchen` als Funktionen definiert.

Tabelle 1.15: Programm: ET-Auszahlung-buchen

```
"Vorgang an Abteilung 2 abgeben"
eval> (ET-Auszahlung-buchen) ==>
"Ist die Kontonummer angegeben? (J/N)"N
"Ist der Betrag groesser gleich 50.00 EUR ? (J/N)"N
"Ist der Zweck angegeben ? (J/N)"N
"Dem Chef vorlegen"
```

Die Lösung im Programm der Tabelle 1.15 S. 83 wirft die Frage auf: Kann eine tiefe Schachtelung von `cond`-Konstrukten durch Nutzung der logischen Verknüpfungen `and` und `or` vermieden werden? Zunächst erläutern wir diese logischen Verknüpfungen, indem wir sie für zwei Argumente (engl.: *binary predicates*) mit dem `cond`-Konstrukt abbilden.

```
eval> (define myAnd
        (lambda (arg_1 arg_2)
          (cond(arg_1 arg_2); 1. Klausel
               (#t #f)      ; 2. Klausel
           )))
eval> (myAnd "Ok?" "Klar!")
        ==> "Klar!"
eval> (myAnd #f "Klar!")
        ==> #f
```

Ist `arg_1` wahr, dann wird die 1. Klausel des `cond`-Konstruktes ausgeführt. Es wird der Wert von `arg_2` festgestellt und als Wert von `myAnd` zurückgegeben. Da `arg_1` wahr ist, entscheidet `arg_2`, ob die `and`-Verknüpfung insgesamt wahr ist. Ist `arg_1` jedoch gleich `false`, also nicht wahr, dann kann sofort (ohne den Wert von `arg_2` nachzufragen) das gesamte `and`-Konstrukt den Wert `#f` erhalten. Analog dazu ist das `or`-Konstrukt für zwei Argumente formulierbar:

```
eval> (define myOr
        (lambda (arg_1 arg_2)
          (cond(arg_1 #t)
               (#t arg_2))))
eval> (myOr "Ok?" "Klar!")
        ==> #t
eval> (myOr #f "Klar!")
        ==> "Klar!"
```

Wir können die Definition von `myOr` noch kürzer formulieren.

```
eval> (define myOr
        (lambda (arg_1 arg_2)
          (cond(arg_1) (arg_2))))
eval> (myOr "Ok?" "Klar!")
        ==> "Ok?"
eval> (myOr #f "Klar!")
        ==> "Klar!"
eval> (myOr #f #f)
```

Bei dieser Definition nutzen wir die Möglichkeit, eine Klausel nur mit dem symbolischen Ausdruck für das Prädikat zu definieren (↪ Abschnitt 1.2.1 S. 53). Ist sein Wert ungleich `false` hat das gesamte `cond`-Konstrukt diesen Wert. Solche Notationen können für „Denkübungen" formuliert werden. Wir sollten jedoch genau prüfen, wann sie im Praxiseinsatz zweckmäßig sind.

Exkurs: `and` (bzw. `or`) und `cand` (bzw. `cor`)

Die obigen Definitionen verweisen auf die in der Informatik gebräuchliche Unterscheidung zwischen `and` (bzw. `or`) und `cand` (bzw. `cor`). Bei `and` (bzw. `or`) sind beide Argumente auszuwerten, auch wenn der logische Wert sich aus dem ersten ergibt (`true` bei `or` bzw. `false` bei `and`). Grund: Mögliche Nebeneffekte der Operanden sollen ausgeführt werden. Bei `cand` und `cor` will man diese möglichen Nebeneffekte vermeiden, z. B. damit folgende Formulierung zu keiner Division durch 0 führt:

```
eval> (define x 0)
eval> (cor (=  0 x) (> (/ 1 x) 7))
        ;bei x = 0 kommt der zweite Teil
        ; nicht zur Ausführung
```

Für die logische Verknüpfung von `a` mit `b` können wir definieren:

```
 cand ≡  (cond (a b) (#t #f))
  and ≡  (cond (a b) (b #f) (#t #f))
  cor ≡  (cond (a #t) (#t b))
   or ≡  (cond (a b #t) (b #t) (#t #f))
```

Die in *PLT-Scheme (Dr. Scheme)* eingebauten `and`- und `or`-Konstrukte entsprechen `cand`- und `cor`-Konstrukten, die nicht auf zwei Argumente beschränkt sind; sie können beliebig viele Argumente haben.

```
eval> (and #t (print "Klar!"))
      ==> "Klar!"
eval> (and #f (print "Klar!"))
      ==> #f
eval> (and 1 2 3 4 5 6)
      ==> 6  ;wahr, da alle Werte
             ; ungleich #f
eval> (and #t (list) (print "Klar?"))
      ==> "Klar?"
eval> (and #f (list) (1 2 3))
      ==> #f ;obwohl das 3. Argument
             ; nicht auswertbar ist,
             ; da 1 keinen Operator
             ; abbilden kann.
eval> (or (or (or 1 #f) (and 2 #f)) 3)
      ==> 1
```

Die Negation eines Wahrheitswertes, das `not`-Konstrukt, lässt sich ebenfalls auf das `cond`-Konstrukt zurückführen:

```
eval> (define myNot
        (lambda (arg)
          (cond(arg #f)
               (#t #t))))
eval> (myNot (and 'Ja '()))
        ==> #f
```

Hinweis: `not` und `null?`
Die leere Liste und der Wahrheitswert `false` sind in *PLT-Scheme (DrScheme)* zu unterscheiden — nicht in allen LISP-Systemen!

```
eval> (null? (list)) ==> #t
eval> (not (list)) ==> #f
```

Im Sinne einer möglichst kleinen Konstrukte-Menge könnten wir auf ein Konstrukt verzichten.

```
eval> (define myNull?
        (lambda (liste)
          (eq? liste (list))))
eval> (myNull? '()) ==> #t
eval> (myNull? 7) ==> #f
eval> (myNull? #t) ==> #f
```

Zur besseren Lesbarkeit verwenden wir trotzdem beide: Das `null?`-Konstrukt zum Prüfen einer leeren Liste und das `not`-Konstrukt, wenn es um den Wahrheitswert `false` geht.

Wie dargelegt, wird in Scheme die Auswertung von `or` beim ersten Wert ungleich `false` beendet, das gesamte `or`-Konstrukt hat dann diesen Wert. Die Auswertung von `and` wird beim ersten Wert `false` beendet, das gesamte `and`-Konstrukt hat `false` als Wert. Das `and`- und das `or`-Konstrukt sind damit noch nicht vollständig spezifiziert. Das `and` und das `or` können in einigen LISP-Systemen, z. B. in Scheme, auch ohne Argument angewendet werden. Das `and` ohne Argument gibt den Wert `#t` zurück, weil kein Argument als kein falsches Argument interpretiert wird. Das `or` ohne Argument führt zu `#f`, da kein Argument als kein wahres Argument interpretiert wird.

```
eval> (and) ==> #t
eval> (or) ==> #f
```

Man könnte spontan mit `and`-Konstrukten die obige begrenzte Eintreffer-Entscheidungstabelle (↪ Tabelle 1.14 S. 82) nach der „Regel-für-Regel-Programmierung“ folgendermaßen notieren:

```
eval> (define  ET-Auszahlung-buchen (lambda ()
        (cond
          ;;Regel 1
          ((and (B1) (B2) (B4)) (A5))
          ;;Regel 2
          ((and (B1) (B2) (B3) (not (B4))) (A1))
          ;;Regel 3
```

```
((and (B1) (B2) (not (B3))
      (not (B4))) (A3) (A5))
;;Regel 4
((and (B1) (not (B2))) (A4))
;;Regel 5
((and (not (B1)) (B3)) (A2))
;;Regel 6
((and (not (B1)) (not (B3))
      (B4)) (A3) (A5))
;;Regel 7
((and (not (B1)) (not (B3)) (not (B4)))
      (A1)))))
```

Die Mängel dieser „Regel-für-Regel-Programmierung“ sind:

1. das Nichtberücksichtigen des aktuellen Wertes einer Bedingung für die Auswahl der nächsten Bedingung und

2. das wiederholte Abfragen von Bedingungen (wiederholtes erzeugen der Nebeneffekte!).

Beispielsweise fragt das obige Konstrukt `ET-Auszahlung-buchen` siebenmal nach der Kontonummer, wenn die Regel R7 zutrifft. Um zumindest nicht jeweils über die Benutzerschnittstelle erneut die Eingaben abzufordern, d. h. die Nebeneffekte der Bedingungen zu vollziehen, könnten die Bedingungen statt als `lambda`-Konstrukte direkt als Wahrheitswert definiert werden:

```
eval> (define B1
        (begin
          (print
            "Ist die Kontonummer angegeben? (J/N)")
          (read)))
        ==>
          "Ist die Kontonummer angegeben? (J/N)"J
```

Damit wird die Eingabe allerdings schon zum Zeitpunkt der Definition der Bedingung gefordert. Eine Zeitpunktverschiebung der Auswertung wäre erforderlich (Näheres ↪ Abschnitt 1.2.3 S. 103).

Für komplexe Fallunterscheidungen gibt es im Allgemeinen weitere eingebaute Konstrukte, die gegenüber geschachtelten `cond`-Konstrukten eine vereinfachte Notation ermöglichen. Ein Beispiel ist das `case`-Konstrukt. Es erlaubt die Angabe von „Treffern“ in einer Liste. Definiert ist es, wie folgt:

```
eval> (case < form > { (< selector_i >
                  { < sexpr_i > }) }) ==>< value >
```

Zunächst wird der Ausdruck $< form >$ ausgewertet. Sein Wert wird sukzessiv mit jedem $< selector_i >$ verglichen, wobei $< selector_i >$ nicht evaluiert wird![11] Die Art des Vergleiches wird vom Typ des $< selector_i >$-Ausdruckes bestimmt. Ist $< selector_i >$ eine Liste, dann wird jedes Element mit dem Wert von $< form >$ verglichen. Ist $< selector_i >$ ein Symbol außer `else`, dann wird die Identität bzw. Wertgleichheit bei Zahlatomen, Strings etc. geprüft. Ist $< selector_i >$ das Symbol `else`, dann ist der Vergleich sofort erfolgreich. Bei einem erfolgreichen Vergleich stoppt der Vergleichsprozess, und die zugehörende Sequenz der $< sexpr_i >$ wird evaluiert. Der Wert des `case`-Konstruktes $< value >$ ist der Wert des zuletzt evaluierten Ausdrucks (entspricht der Definition des `cond`-Konstruktes).

Beispiel: `case`**- und** `cond`**-Konstrukt**

```
eval> (define Zustimmung?
        (lambda ()
          (case (read)
          ((J Ja Jawohl Jawoll Y Yes Klar
            Zustimmung Gewiss sicher freilich
            selbstredend selbstverstaendlich
            allerdings gern X T true oui quit si)
           #t)
          (else #f))))
eval>(Zustimmung?)
     Jawohl ;Eingabe
     ==> #t ;Wert

eval> (define Zustimmung?
        (lambda ()
         (let ((Antwort (read)))
           (cond((or (eq? Antwort 'J)
                     (eq? Antwort 'Ja)
                     (eq? Antwort 'Jawohl)
                          .
                          .
                          .
                     (eq? Antwort 'Si)) #t)
                (#t #f)))))
```

Hinweis: `case`- und `cond`-Konstrukt
Beide Konstrukte basieren auf dem `if`-Konstrukt. Sie sind eine syntaktische Vereinfachung. In *PC Scheme* beispielsweise zeigen die folgenden `EXPAND-MACRO`-Anwendungen, wie `case` und `cond` dort als Makros definiert sind (Näheres zu Makros ↪ Abschnitt 2.5.3 S. 293)

[11] Das `case`-Konstrukt heißt deshalb konsequenterweise bei einigen LISP-Systemen `SELECTQ` (z. B. in *TLC-LISP*), wobei `Q` für `QUOTE` steht.

```
PC-Scheme-eval> (EXPAND-MACRO
                  '(case (read) ((J Ja) 1 2)
                     (else 3 4)))
                  ==>
 ((lambda (%00000)
   (if (memv %00000 '(J Ja));Das memv-Konstrukt
       (begin 1 2)       ; prüft ob der Wert von
       (begin 3 4)))     ; %00000 Element der
 (read))                 ; Liste (J Ja) ist.

PC-Scheme-eval> (EXPAND-MACRO
                  '(cond((eq? 'J (read)) 1 2)
                           (#t 3 4))) ==>
        (if (eq? 'J (read))
            (begin 1 2)
            (cond(#t 3 4)))
```

Iteration

Eine Iteration (auch Repetition genannt) ermöglicht das wiederholte Durchlaufen bzw. Anwenden eines Konstruktes. Die Abbildung 1.18 S. 90 zeigt die grafische Darstellung einer Iteration in *Jackson-Notation* (↪ [98]). Der Modul `Gesamt` besteht aus einer beliebig langen Folge von Moduln `Wiederholungs-Teil`, daher notiert mit einem Stern (↪ Abbildung 1.18 S. 90). Dabei ist zu beachten:

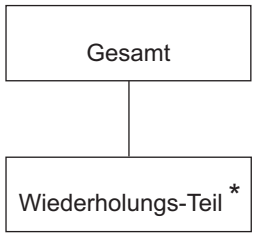
1. `Gesamt` ist die Iteration und
2. die Häufigkeit des Auftretens von `Wiederholungs-Teil` ist eine Eigenschaft von `Gesamt` und nicht von `Wiederholungs-Teil`.

Der Stern in der rechten oberen Ecke dokumentiert das mehrfache Auftreten von `Wiederholungs-Teil` innerhalb von `Gesamt`. Eine Iteration wird beendet:

1. entweder durch Zutreffen einer Abbruchbedingung, wobei die Ereignisprüfung vor oder nach einem Durchlauf erfolgen kann,
2. oder nach Vollzug einer vor Beginn der Iteration festgelegten Anzahl von Durchläufen.

In vielen Programmiersprachen werden zur Kennzeichnung einer Iteration Bezeichner wie `do`, `while`, `repeat`, `until` oder `for` benutzt. In LISP lassen sich entsprechende Konstrukte definieren, falls diese im jeweiligen LISP-System nicht als eingebaute Konstrukte zur Verfügung stehen.

Charakteristisch für LISP ist die Konstruktion einer Iteration in Form einer rekursiven Definition. Iterationen können als Spezialfälle der Re-



Legende:
M. A. Jackson-Notation (↪ [98])

Abbildung 1.18: Grafische Darstellung einer Iteration

kursion betrachtet werden. Rekursive Darstellungen sind für LISP prägend, z. B. sind sowohl die Syntax als auch die Semantik der symbolischen Ausdrücke rekursiv definiert (↪ Abschnitte 1.1.1 S. 15 und 1.1.2 S. 20). Wegen ihrer großen Bedeutung ist der Rekursion in diesem Kapitel ein eigener Abschnitt gewidmet.

In LISP unterscheiden wir im Zusammenhang mit Wiederholungsstrukturen drei Abbildungskonzepte:

1. die üblichen Iterationskonstrukte, z. B. das `do`-Konstrukt,

2. die Anwendung von `map`-Konstrukten (*mapping functions*) und

3. die Iteration als Spezialfall der Rekursion.

Zunächst behandeln wir als bedingte Wiederholung das `do`-Konstrukt. Nach diesem vielfältig nutzbaren Iterationskonstrukt wird die „klassische“ Iteration mit `map`-Konstrukten erläutert. Die `map`-Konstrukte realisieren die Iteration, indem sie eine Funktion wiederholt auf eine Liste von Argumenten anwenden.

Das `do`-Konstrukt gliedert sich in drei Definitionsbereiche:

```
eval> (do (Zustandsvariablen)
          (Endebehandlung)
           Verarbeitungskörper)
```

1. *Zustandsvariablen:*
   Die Definition einer Zustandsvariablen gliedert sich in die folgenden drei Teile:

   (a) $<$ *variable* $>\equiv$ Namen der Variablen

(b) $<$ *init_sexpr* $>\equiv$ Symbolischer Ausdruck, an dessen Wert die $<$ *variable* $>$ zu Beginn der Iteration gebunden wird. Es handelt sich um den Initialwert für die $<$ *variable* $>$.

(c) $<$ *iter_sexpr* $>\equiv$ Symbolischer Ausdruck, der bei jeder Wiederholung evaluiert wird und an dessen Wert $<$ *variable* $>$ gebunden wird. Es handelt sich um den Iterationswert (Inkrement oder Dekrement) für die $<$ *variable* $>$.

2. *Endebehandlung:*
   Die Definition der Endebehandlung entspricht einer Klausel des `cond`-Konstruktes: Zuerst wird die Abbruchbedingung $<$ *exit*? $>$ evaluiert und, falls ihr Wert ungleich `false` ist, wird die Sequenz von symbolischen Ausdrücken $<$ *exit_sexpr*$_1$ $>$ ... $<$ *exit_sexpr*$_n$ $>$ der Reihe nach von links nach rechts evaluiert. Der zuletzt ermittelte Wert ist der Wert des gesamten `do`-Konstruktes.

3. *Verarbeitungskörper:*
   Eine Sequenz von symbolischen Ausdrücken, $<$ *body_sexpr*$_1$ $>$ ... $<$ *body_sexpr*$_m$ $>$, die der Reihe nach ausgewertet werden. Sie dienen auch zur Manipulation der Variablen und des Abbruchtestes. Der Rückgabewert des `do`-Konstruktes wird nur durch die Endebehandlung definiert.

Die Definitionsbereiche müssen nicht zwingend einen symbolischen Ausdruck aufweisen. Das `do`-Konstrukt ist, wie folgt, definiert:

`eval>` `(do (` { ($<$ *variable* $>$ {$<$ *init_sexpr* $>$ {$<$ *iter_sexpr* $>$}}) } `)`
`(`$<$ *exit*? $>$ {$<$ *exit_sexpr* $>$} `)`
{$<$ *body_sexpr* $>$} `)`

Anhand von zwei Beispielen zeigen wir die Arbeitsweise und Wirkung des `do`-Konstruktes. Zunächst bilden wir eine Zählschleife ab, indem wir den Satz `LISP hat ein dia-logisches Konzept` siebenmal schreiben. Danach summieren wir alle Elemente einer Liste, vorausgesetzt, es handelt sich um Zahlen.

Beispiel: Siebenmal dieselbe Zeichenkette ausgeben

Lösung 1: Zählschleife mit Verarbeitung im $<$ *body_sexpr* $>$

```
eval> (do ((Zaehler 7 (- Zaehler 1)))
          ((= 0 Zaehler))
          (print
            "LISP hat ein dia-logisches Konzept!")
          (newline)) ==>
"LISP hat ein dia-logisches Konzept!"
"LISP hat ein dia-logisches Konzept!"
```

```
"LISP hat ein dia-logisches Konzept!"
"LISP hat ein dia-logisches Konzept!"
"LISP hat ein dia-logisches Konzept!"
"LISP hat ein dia-logisches Konzept!"
"LISP hat ein dia-logisches Konzept!"
eval>
```

Lösung 2: Zählschleife mit Verarbeitung im $<$ *iter_sexpr* $>$

```
eval> (do ((Zaehler 7 (begin
                        (print
   "LISP hat ein dia-logisches Konzept!")
                        (newline)
                        (- Zaehler 1))))
           ((= 0 Zaehler)))
"LISP hat ein dia-logisches Konzept!"
"LISP hat ein dia-logisches Konzept!"
"LISP hat ein dia-logisches Konzept!"
"LISP hat ein dia-logisches Konzept!"
"LISP hat ein dia-logisches Konzept!"
"LISP hat ein dia-logisches Konzept!"
"LISP hat ein dia-logisches Konzept!"
```

Lösung 3: Zählschleife mit Verarbeitung im $<$ *exit*? $>$**-Ausdruck**

```
eval> (do ((Zaehler 6 (- Zaehler 1)))
          ((begin (print
   "LISP hat ein dia-logisches Konzept!")
                  (newline)
                  (= 0 Zaehler)))) ==>
"LISP hat ein dia-logisches Konzept!"
"LISP hat ein dia-logisches Konzept!"
"LISP hat ein dia-logisches Konzept!"
"LISP hat ein dia-logisches Konzept!"
"LISP hat ein dia-logisches Konzept!"
"LISP hat ein dia-logisches Konzept!"
"LISP hat ein dia-logisches Konzept!"
```

Die Lösungen 2 und 3 stellen eine „trickreiche“ Nutzung des `do`-Konstruktes dar. Sie sind weniger transparent als die Lösung 1. Hier sind sie nur zum Verstehen des `do`-Konstruktes formuliert. Unstrittig ist die Lösung 1 den beiden Lösungen 2 und 3 vorzuziehen. Sie hat die vom Leser (gewohnte) einfache `do`-Struktur und wird daher schneller verstanden.

Beispiel: Zahlen in einer Liste summieren

Lösung 1: Addieren der Zahlen im Verarbeitungskörper des `do`-Konstruktes

```
eval> (define Summe-der-Zahlen (lambda (l)
        (do ((liste l (cdr liste))
             (Akkumulator 0)) ;Hilfs-
               ; variable für das Ergebnis
            ((null? liste) Akkumulator) ;Ende-
                                        ; bedingung
          (cond((number? (car liste))
                (set! Akkumulator
                       (+ Akkumulator (car liste))))
               (#t "Naechstes Listenelement")))))
eval>  (Summe-der-Zahlen '(A 3 B 1 C 2)) ==> 6
```

Lösung 2: Addieren der Zahlen im Iterationsteil des `do`-Konstruktes

```
eval> (define Summe-der-Zahlen (lambda (l)
        (do ((liste l (cdr liste))
             (Akkumulator 0)) ; Hilfs-
               ; variable für das Ergebnis
            ((null? liste) Akkumulator) ;Ende-
                                        ; bedingung
          (cond((number? (car liste))
                (set! Akkumulator
                       (+ Akkumulator (car liste))))
               (#t "Naechstes Listenelement")))))
eval> (Summe-der-Zahlen '(1 2 3 4 A B C D)) ==> 10
```

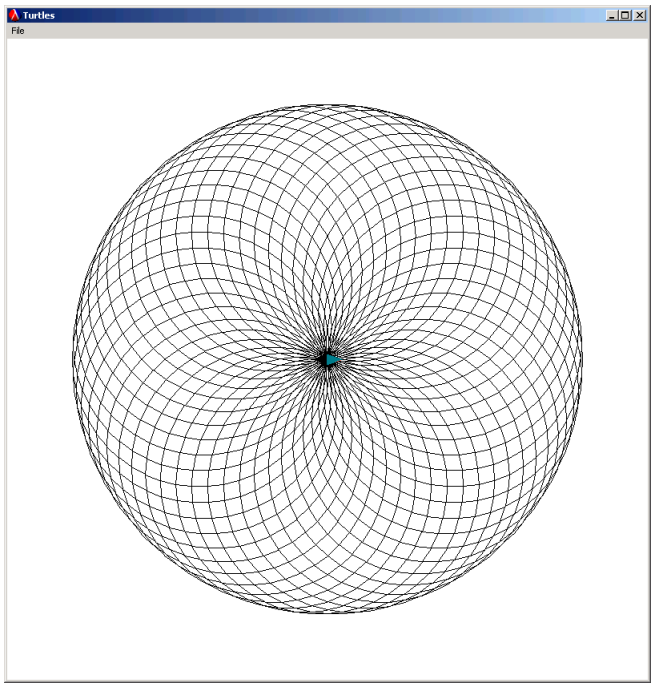
Hinweis: `do`-Implementation.
Anhand der `do`-Implementation wird deutlich, dass die Iteration mit dem `do`-Konstrukt sich auf die Rekursion mittels `letrec` abstützt. Wie das `let`-Konstrukt (↪ S. 72), so ist auch das `do`-Konstrukt in *PC Scheme* als Makro-Konstrukt realisiert. Analog zum `begin0`-Makro (↪ Abschnitt 13 S. 78) wird auch hier eine lokale Variable generiert. Mit Hilfe des `EXPAND-MACRO`-Konstruktes ist das Ergebnis des expandierens darstellbar. Dabei formulieren wir die $<$ *sexprs* $>$ der obigen `do`-Definition als Zeichenketten:

```
PC-Scheme-eval>
 (define Foo
   (EXPAND-MACRO
        '(do ((V-1 "<v-1_init_sexpr>"
                   "<v-1_iter_sexpr>")
              (V-2 "<v-2_init_sexpr>"
                   "<v-2_iter_sexpr>"))
             ((or (eq? V-1 'ENDE)
                  (eq? V-2 'ENDE))
               "<exit_sexpr_a>"
               "<exit_sexpr_b>")
          "<body_sexpr>")))

PC-Scheme-eval>
```

```
  FOO ==>
      ((Letrec
          ((%00000
            (lambda (V-1 V-2)
              (if (or (eq? V-1 'ENDE)
                      (eq? V-2 'ENDE))
                  (begin "<exit_sexpr_a>"
                         "<exit_sexpr_b>")
                  (begin "<body_sexpr>"
                    (%00000
                      "<v-1_iter_sexpr>"
                      "<v-2_iter_sexpr>"))))))
         %00000)
        "<v-1_init_sexpr>"
        "<v-2_init_sexpr>")
```

Zum Verständnistraining expandieren wir die Lösungen 1 und 3 des obigen Beispiels „Siebenmal eine konstante Zeichenkette ausgeben“:

```
PC-Scheme-eval>
  (EXPAND-MACRO
        '(do ((Zaehler 7 (- Zaehler 1)))
             ((= 0 Zaehler))
             (print
  "LISP hat ein dia-logisches Konzept!")
             (newline)
         )) ==>
      ((letrec
          ((%00000
            (lambda (Zaehler)
              (let ((%00001 (= 0 Zaehler)))
                (if %00001
                    %00001
                    (begin
                      (print
        "LISP hat ein dia-logisches Konzept!")
                      (newline)
                      (%00000
                        (- Zaehler 1))))))))
         %00000) 7)
PC-Scheme-eval>
  (EXPAND-MACRO
        '(do ((Zaehler 6 (- Zaehler 1)))
             ((begin (print
        "LISP hat ein dia-logisches Konzept!")
             (= 0 Zaehler)))
         )) ==>
      ((letrec
          ((%00000
            (lambda (Zaehler)
```



Legende:
Turtle-Grafik ↪ Abschnitt A.3.2 S. 475; gezeichnte mit den Konstrukten `(draw` < *strich-laenge* >`)` und `(turn` < *winkel* >`)` ↪ Programm 1.16 S. 97

Abbildung 1.19: *Turtle*-Grafik: `nm-Eck`

```
      (let ((%00001
             (begin
               (print
"LISP hat ein dia-logisches Konzept!")
               (newline)
               (= 0 Zaehler))))
        (if %00001
            %00001
            (begin
              (%00000
                (- Zaehler 1))))))))
 %00000) 6)
```

Beispiel: Geschachtelte Iterationen

Erfahrungsgemäß wird ein Verstehen von Iterationkonstrukten schwieriger, wenn sie geschachtelt vorkommen, d. h. wenn Iterationen innerhalb

von anderen Iterationen stattfinden. Als grobe Daumenregel ist ein intuitives *Ranking* der Verstehbarkeit von Konstrukten annehmbar; von leicht mit wachsender Schwierigkeit wie folgt:

1. *Sequenz* z. B. `(begin ...)` ↪ S. 77
2. *Alternative* z. B. `(cond ...)` ↪ S. 53
3. *Iteration* z. B. `(do ...)` ↪ S. 91
4. (mehrfach) *geschachtelte Iterationen* z. B. `(do ... (do ... ))` ↪ S. 97
5. *Sprünge* (nicht lineare Kontrollstrukturen) z. B. `(call/cc ...)` ↪ S. 285
6. *Rekursion* z. B. `(letrec ...)` ↪ S. 76
7. *„geschachtelte" Rekursionen* z. B. `(Y ...)` ↪ S. 277
8. *Nebenläufigkeit* z. B. `(thread ...)` ↪ S. 355

Das Konstrukt `nm-Eck` (↪ S. 97) zeichnet mit Hilfe der *Turtle*-Grafik (↪ Abschnitt A.3.2 S. 475) ein Bild, das einer sogenannten *Hilbert*-Kurve[12] ähnelt. Die Iterationen im Konstrukt `n-Eck` befindet sich in der Iteration des Konstruktes `nm-Eck` und zwar aufgrund der Applikation von `n-Eck` innerhalb von `mn-Eck`. Die komplex wirkende Grafik (↪ Abbildung 1.19 S. 95) basiert daher auf zwei ineinander geschachtelten Iterationen.

Lineare Konstrollstrukturen

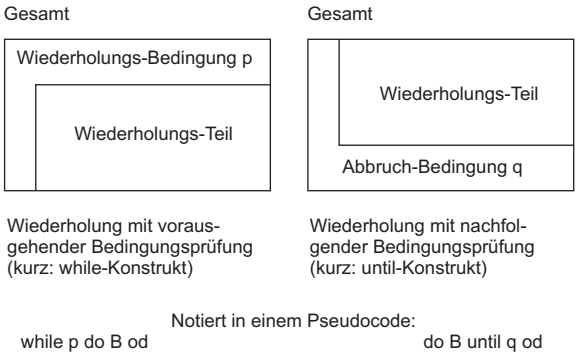
Im Rahmen des Entwurfes von linearen Kontrollstrukturen (↪ Exkurs S. 98) werden Iterationen oft in der von *I. Nassi* und *B. Shneiderman* (↪ [138]) vorgeschlagenen Form dargestellt (↪ Abbildung 1.20 S. 98). Die Iteration mit der Bedingungsprüfung vor jedem Wiederholungsdurchlauf wird auch abweisende Schleife oder salopp: „kopfgesteuerte" Schleife genannt. Sie ist zu unterscheiden von der Iteration mit der Bedingungsprüfung nach jedem Wiederholungsdurchlauf. Diese wird als nichtabweisende Schleife oder salopp: „fußgesteuerte" Schleife bezeichnet. Im ersten Fall, dem `while`-Konstrukt, wird so lange wiederholt, bis die Bedingung erfüllt ist. Im zweiten Fall, dem `until`-Konstrukt, wird der Wiederholungsvorgang beendet, wenn die Bedingung erfüllt ist.

[12] Der Mathematiker *David Hilbert* (⋆ 23.01.1862, Königsberg; † 14.02.1943, Göttingen) hat eine stetige Kurve entdeckt, die **durch Wiederholung ihres Konstruktionsverfahrens** jedem beliebigen Punkt einer quadratischen Fläche beliebig nahe kommt und die Fläche vollständig ausfüllt. Im übertragenen Sinne bildet das `mn-Eck` eine *Hilbert*-Kurve ab.

```
eval> ;Bereitstellung der
      ; klassischen Turtle-Grafik
      (require graphics/turtles)
eval> ;Iteration
      (define n-Eck
       (lambda (n l)
        (do ((i 0 (+ i 1)))
            ((= i n))
            (turn (/ 360 n))
            (draw l))))
eval> ;Iteration innerhalb
      ; einer Iteration
     (define mn-Eck
      (lambda (n l)
       (do ((i 0 (+ i 1)))
           ((= i n))
           (turn (/ 360 n))
           (n-Eck n l))))
eval> ;Grafik-Fenster
      (turtles) ==> #(struct:object:canvas% ...)
eval> (mn-Eck 50 20) ==>
      ;Hilbert-Kurve
```

Legende:
Ergebnis ↪ Abbildung 1.19 S. 95; *Turtle*-Grafik ↪ Abschnitt A.3.2 S. 475.

Tabelle 1.16: Programm `nm-Eck`



Legende:
Grafische Darstellung nach DIN 66261 (↪ [52]) — auch als Struktogramm oder Nassi/Shneiderman-Diagram bezeichnet (↪ [138]).

Abbildung 1.20: Bedingte Schleifen

Exkurs: *Lineare Struktur*
Die Bezeichnung „*Lineare Struktur*“ wird primär im imperativ-geprägten Paradigma (↪ S. 141) verwendet. Da besteht eine lineare (Kontroll-)-Struktur aus „Bausteinen“, die nur einen „Eingang“ und einen „Ausgang“ für den „Kontrollfluss“ (bzw. „Datenfluss“) haben. Beispielsweise macht ein „Baustein“, der mehrere Konstrukte in der Art von `break` oder `exit` aufweist, die Struktur zu einer nicht-linearen Struktur.

In den beiden folgenden Beispielen wird die Situationsänderung für die Bedingungsprüfung über Zufallszahlen bewirkt. Das `random`-Konstrukt gibt als Wert Pseudozufallszahlen zurück und zwar in dem Wertebereich von Null bis Wert des Argumentes minus 1.

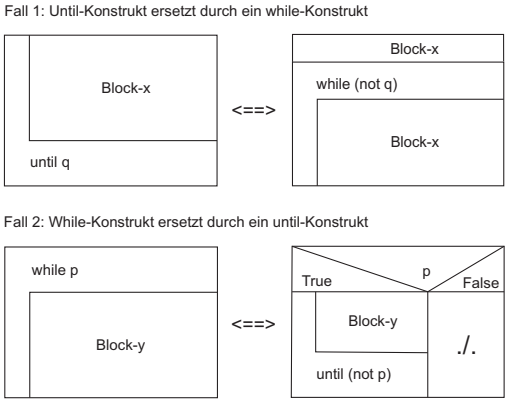
```
eval> (random 5) ==> 3 ;0 <= Pseudozufallszahl < 5
```

Beispiele: Bedingte Schleifen (`while`-/`until`-Konstrukte)

```
eval> (define Block
        (lambda () (print "Block verarbeitet")
                   (newline)))
eval> (define Wiederholungs-Bedingung
        (lambda () (< (random 5) 3)))
eval> (define Abbruch-Bedingung
        (lambda () (not
                     (Wiederholungs-Bedingung))))

eval> (define While-Beispiel
        (lambda ()
          (do ((p ;Bedingung
```

```
                ;do-Initialisierung
                (Wiederholungs-Bedingung)

                ;do-Iterationswert
                (Wiederholungs-Bedingung)))

              ((not p)) ;do-Exit-Bedingung
              (Block))))

eval> (While-Beispiel) ==>
      "Block verarbeitet"
      "Block verarbeitet"
eval> (While-Beispiel) ==>
      "Block verarbeitet"
      "Block verarbeitet"
      "Block verarbeitet"
      "Block verarbeitet"

eval> (define Until-Beispiel
        (lambda ()
          (do ((q
                #f  ;Um Block zumindest
                    ; einmal abzuarbeiten.
                (Abbruch-Bedingung)))
              (q)    ; do-Exit-Bedingung
              (Block))))
eval> (Until-Beispiel) ==>
      "Block verarbeitet"
eval> (Until-Beispiel) ==>
      "Block verarbeitet"
      "Block verarbeitet"

eval> (set! Wiederholungs-Bedingung
        (lambda () #f))
eval> (While-Beispiel) ==> ;Keine Ausgabe
eval> (Until-Beispiel) ==>
      "Block verarbeitet"
```

Das obige Konstrukt `Until-Beispiel` entspricht einer Sequenz aus `Block` und anschließendem `while`-Konstrukt. Die Beziehungen zwischen `while`- und `until`-Konstrukten zeigt Abbildung 1.21 S. 100. Aufgrund der Listen-Dominanz bezieht sich in LISP die Wiederholung häufig auf die Abarbeitung aller Elemente einer Liste. Ist eine Funktion auf jedes Element einer Liste anzuwenden, dann bilden wir diese Iteration mit einem `map`-Konstrukt („*mapping function*“) ab. LISP-Systeme haben ein ganzes Bündel von *mapping functions*. Einige sammeln die Werte der wiederholten Funktionsapplikation und geben diese als Wert zurück, andere führen die Funktionsapplikation nur zur Erzielung von Nebeneffek-



Legende:
↪ Abbildung 1.20 S. 98
./. ≡ Identitätsfunktion (zweckmäßig zur Kennzeichnung, dass eine Blockeintragung nicht vergessen wurde).

Abbildung 1.21: Iterationskonstrukte

ten durch; ihr Rückgabewert ist unerheblich.

`eval>` $(< mapping\_function > < user\_function > < arguments >)$
`==>` $< return\_value >$

mit:

$< return\_value >$ ≡ Abhängig von der jeweiligen $< mapping\_function >$ entsteht ein Rückgabewert. Stets wird die Applikation der $< user\_function >$ auf die einzelnen Argumente von $< arguments >$ durchgeführt:

$$(< user\_function >< argument\_1 >) \texttt{ ==>} < value_1 >$$
$$\vdots$$
$$(< user\_function >< argument\_n >) \texttt{ ==>} < value_n >$$

Gegebenenfalls wird anschließend aus $< value_1 >$ bis $< value_n >$ der Rückgabewert gebildet, z. B. eine Liste.

Bei den `map`-Konstrukten besteht die eigentliche Iteration in der Wiederholung der Anwendung von $< user\_function >$ auf die einzelnen Argumente. Dabei ist $< user\_function >$ eine Funktion mit einer `lambda`-Variablen, d. h. eine Funktion definiert für ein Argument (engl.: *unary function*). Da das `map`-Konstrukt diese $< user\_function >$ als Argument hat, ist es selbst eine Funktion höherer Ordnung. Eine Funktion höherer

Ordnung (engl.: *higher-order function*) ist eine Funktion, die entweder eine Funktion als Argument hat und/oder eine Funktion als (Rückgabe-)Wert. Der Vorteil dieser `map`-Iteration ist das Abstrahieren von den Details, wie die einzelnen Argumente der Reihe nach selektiert werden und die Abbruchbedingung geprüft wird.

Das folgende Beispiel Sachmittelplanung einer Behörde (↪ S. 101) demonstriert die beiden *mapping functions*: Das `map`-Konstrukt und das `for-each`-Konstrukt.

```
eval> (map list '(1 2 3)) ==> ((1) (2) (3))
eval> (for-each print '(1 2 3)) ==> 123
```

Beide wenden eine Funktion, die für ein Argument definiert ist, auf eine Liste von Argumenten an. Der Wert des `map`-Konstruktes ist eine Liste, die die Resultate der einzelnen Funktionsanwendungen enthält. Der Wert des `for-each`-Konstruktes ist undefiniert, denn es dient zur Ausführung von Nebeneffekten.

Hinweis: *Mapping functions*.
In Common LISP umfasst das Repertoire an *mapping functions* auch Konstrukte, bei den sich die Funktionsapplikation auf die jeweilige Restliste bezieht, z. B. `maplist` oder `mapcon`. Die beiden hier verwendeten *mapping functions* sind in Scheme-Notation, wie folgt, definierbar (Näheres zur Rekursion ↪ Abschnitt 1.3 S. 113):

```
eval> (define for-each (lambda (function liste)
        (cond((null? liste) (void))
              (#t (function (car liste))
                 (for-each function
                           (cdr liste))))))

eval> (for-each print '(1 2 3)) ==> 123

eval> (define map (lambda (function liste)
        (cond((null? liste) (list))
              (#t (cons (function (car liste))
                        (map function
                             (cdr liste)))))))

eval> (map list '(1 2 3)) ==> ((1) (2) (3))
```

Beispiel: Sachmittelbedarf

Wir betrachten eine Behörde und deren Sachmittelbedarf. Für das Jahr 2010 hatte die Abteilung `W` einen Bedarf pro Quartal in folgender Höhe:

```
eval> (define Sachmittel-Quartalsbedarf-W-2010
         '(150000 100000 80000 120000))
```

Für das Jahr 2011 ist eine Steigerung von 2,45 Prozent vorzusehen, d. h. jeder Quartalsbetrag ist mit 1.0245 zu multiplizieren, um die Werte für 2011 zu erhalten. Die Berechnung der neuen gerundeten Werte kann als wiederholte Anwendung der anonymen Funktion:

```
eval> (lambda (Betrag)
        (round (* 1.0245 Betrag)))
      ==> #<procedure>
```

definiert werden.

```
eval> (define Sachmittel-Quartalsbedarf-W-2011
       (map (lambda
              (Betrag) (round (* 1.0245 Betrag)))
            Sachmittel-Quartalsbedarf-W-2010))

eval> Sachmittel-Quartalsbedarf-W-2011
        ==> (153675.0 102450.0 81960.0 122940.0)
```

Wir nehmen nun an, dass die Quartalswerte für 2010 für alle Abteilungen der Behörde in folgender Matrix-Struktur vorliegen:

```
eval>
(define Sachmittel-Quartalsbedarf-2010
  '((Leitung        ( 10000  10000  10000  10000))
    (Abteilung-E    (450000 423000 411000 567230))
    (Abteilung-T    (159000 156000  85000 121300))
    (Abteilung-W    (150000 100000  80000 120000))
    (Stelle-Hamburg ( 34500  32900  25000  55000))
   ))
```

Zur Berechnung der Werte für das Jahr 2011 sind zwei geschachtelte Iterationen notwendig. Es ist über die 5 Zeilen und die 4 Betragsspalten der Matrix-Struktur zu iterieren.

```
eval> (define Sachmittel-Quartalsbedarf-2011
       (map (lambda (Organisation)
              (list (car Organisation)
                    (map (lambda (Betrag)
                           (round (* 1.0245 Betrag)))
                         (car (cdr Organisation)))))
            Sachmittel-Quartalsbedarf-2010))

eval>
Sachmittel-Quartalsbedarf-2011 ==>
 ((Leitung (10245.0 10245.0 10245.0 10245.0))
  (Abteilung-E (461025.0 433364.0 421070.0 581127.0))
  (Abteilung-T (162896.0 159822.0 87082.0 124272.0))
  (Abteilung-W (153675.0 102450.0 81960.0 122940.0))
  (Stelle-Hamburg (35345.0 33706.0 25612.0 56348.0)))
```

Um den Gesamtbedarf an Sachmitteln für die Behörde für das Jahr 2011 zu berechnen, sind alle Beträge in der Matrix-Struktur zu summieren. Dazu ist in den geschachtelten Iterationen ein Akkumulator erforderlich. Dieser ist als lokale Variable mit Hilfe des `let`-Konstruktes definierbar. Da der Endwert des Akkumulators die gesuchte Lösung darstellt, sind die Rückgabewerte der beiden `map`-Konstrukte nutzlos. Wir ersetzen daher die `map`-Konstrukte durch `for-each`-Konstrukte. Das `for-each`-Konstrukt unterscheidet sich von dem `map`-Konstrukt dadurch, dass die Iterationsergebnisse nicht den Rückgabewert bilden.

```
eval> (define Gesamtbedarf-2011
       (let ((Akkumulator 0))
         (for-each
            (lambda (Organisation)
              (for-each
                 (lambda (Betrag)
                   (set! Akkumulator
                         (+ Akkumulator Betrag)))
                 (car (cdr Organisation))))
             Sachmittel-Quartalsbedarf-2011)
          Akkumulator))

eval> Gesamtbedarf-2011 ==> 3083674.0
```

1.2.3 Zeitpunkt einer Wertberechnung

Im Rahmen der Selektion (↪ S. 80) wurde anhand einer Entscheidungstabelle (↪ Tabelle 1.14 S. 82) deutlich, dass bei Nebeneffekten der Zeitpunkt einer Wertberechnung bedeutsam ist. Wir wollen explizit den Zeitpunkt festlegen, wann die einzelnen Bedingungen ihren Wert berechnen. Zum Zeitpunkt der Definitionen von B1, B2, B3 und B4 dürfen die Abfragen, d. h. die Nebeneffekte durch die `print-` und `read`-Konstrukte, noch nicht zur Ausführung kommen. Wir haben daher die Bedingungen als Funktionen definiert und ihren Berechnungszeitpunkt durch den Funktionsaufruf terminiert (↪ Programm in Tabelle 1.15 S. 83). Die Wertberechnung, hier die Abfrage, wurde auf den Applikationszeitpunkt verschoben.

Zur Verschiebung des Zeitpunktes der Wertberechnung eines beliebigen symbolischen Ausdrucks $< sexpr >$ notieren wir diesen $< sexpr >$ innerhalb eines `lambda`-Konstruktes, das keine `lambda`-Variablen besitzt. Mit dem Evaluieren dieses `lambda`-Konstruktes entsteht eine Funktion, kurz „*thunk*“ genannt.

Hinweis: *Thunk*
Eine Funktion, definiert für kein Argument, wird als *thunk* bezeichnet. Dieses Wort stammt aus der ALGOL-Welt.[13] Welche ursprüngliche Bedeu-

[13] ALGOL (*Algorithmic Language*) bezeichnet eine Familie von Programmiersprachen, die

tung es hatte ist unbekannt. Es gibt nur scherzhafte Interpretationen, wie den Verweis auf „Registergeräusche“.

Zum gewünschten Zeitpunkt applizieren wir *thunk* (unsere Funktion) und veranlassen damit die Wertberechnung. Die Kombination von Funktionsdefinition (`lambda`-Konstrukt) und Funktions-Applikation (implizites oder explizites `apply`-Konstrukt) bewirkt die Zeitpunktverschiebung der Wertberechnung.

```
eval> ;;<value>, also der Wert von <sexpr>,
      ;; wird jetzt berechnet.
      (define Foo <sexpr>)

eval> ;;Der berechnete Wert wird jetzt
      ;; nur gesucht und ausgegeben
      Foo ==> <value>

eval> ;;Der Wert des lambda-Konstruktes
      ;; wird berechnet.
      ;; <value>, also der Wert von <sexpr>,
      ;; ist noch nicht berechnet.
      (define Foo (lambda () <sexpr>))

eval> ;;<value> wird berechnet
      ;;  und ausgegeben.
      (Foo) ==> <value>

eval> ;;<value> wird berechnet
      ;;  und ausgegeben.
      (apply Foo null) ==> <value>
```

Potentiell besteht mit der Kombination `quote` und `eval` eine Alternative zur Kombination `lambda` und `apply`. Die Auswertung des symbolischen Ausdrucks $<$ *sexpr* $>$ wird durch das `quote`-Konstrukt blokkiert, so dass sich $<$ *sexpr* $>$ wie eine Konstante verhält, die mit dem `eval`-Konstrukt zum gewünschten Zeitpunkt aktiviert wird (↪ Abschnitt 1.1.3 S. 27).

```
eval> ;;<value>, also der Wert von <sexpr>,
      ;; wird nicht berechnet. Die Auswertung
      ;; von <sexpr> ist blockiert.
      (define Foo '<sexpr>)

eval> ;;Der Wert von FOO, also <sexpr>,
      ;; wird ausgewertet.
      (eval Foo) ==> <value>
```

Exkurs: Zwischenschritt *Compilieren*
Für die Festlegung des Zeitpunktes einer Werteberechnung könnte auch

≈ 1950 . . . 1985 eingesetzt wurde. Die ALGOL-Varianten beeinflussten viele Programmiersprachen, z. B. die Sprache *Pascal*.

der Zwischenschritt *Compilieren* genutzt werden, wie das folgende Beispiel skizziert:

```
eval> (define Foo ((lambda (x) (* x x)) 2))
eval> Foo ==> 4
eval> (define Foo (compile ((lambda (x) (* x x)) 2)))
eval> (compiled-expression? Foo)
      ==> #t
eval> Foo ==> ...; Hexadezimalcode
eval> (eval Foo) ==> 4
```

Im Folgenden sind beide Möglichkeiten für die Entscheidungstabelle von Abbildung 1.14 S. 82 dargestellt. Das erste Programm (↪ Tabelle 1.17 S. 106) nutzt den funktionalen Ansatz. Das zweite Programm (↪ Tabelle 1.18 S. 107) verwendet die Auswertungsblockierung in Verbindung mit dem `eval`-Konstrukt.

Die Benutzerschnittstelle ist in der ET-Abbildung des Abschnittes 1.1.2 (↪ Programm in Tabelle 1.15 S. 83) außerhalb des ET-Konstruktes definiert. Hier sind in beiden Fällen die Bedingungen und Aktionen mit Hilfe des `let`-Konstruktes als lokale Variable des Konstruktes `ET-Auszahlung-buchen` formuliert, also innerhalb der ET definiert. Damit wird einerseits ein ungewolltes Überschreiben im Falle bestehender oder neuer namensgleicher Konstrukte vermieden und andererseits die Zugehörigkeit zum ET-Konstrukt unmittelbar sichtbar. Zusätzlich bewahren wir uns dadurch die leichtere Überschaubarkeit der *top level*-Umgebung, weil diese dann weniger Einträge hat.

1.2.4 Typische Fehler

Ob LISP-Anfänger oder LISP-Profi („LISP-wizard"), alle machen Fehler, der eine mehr, der andere weniger. Missverständliche Vorgaben, Unkenntnis der Zusammenhänge, logische Fehlschlüsse und Flüchtigkeit sind häufige Ursachen. Nicht selten führt daher erst die Fehlersuche und Fehlerkorrektur zu einem tieferen Verständnis des Problems und seiner Lösung. Die folgenden fehlerhaften Konstrukte bieten daher die Möglichkeit, die bisher skizzierte LISP-Welt nochmals zu durchdenken.

Es handelt sich um klassische Anfängerfehler, die gleichermaßen als Flüchtigkeitsfehler den LISP-Profis unterlaufen. In der Praxis wird jeder unter derartigen „Denkbeulen" selbst leiden müssen, ehe er die Souveränität gewinnt, um solche Fehler spontan zu erkennen und lächelnd beseitigen zu können.

Fehler bei der Funktionsdefinition

Gewünschtes Konstrukt

```
eval> (define Verdopplung
        (lambda (Zahl) (* Zahl 2)))
```

```
;;;;Begrenzte Eintreffer-ET mit lokalen Fuktionen
eval> (define ET-Auszahlung-buchen (let
     ((B1 (lambda ()
            (print "Ist die Kontonummer angegeben? (J/N)")
            (read)))
      (B2 (lambda ()
            (print
      "Ist die Kontonummer kleiner gleich 40000 ? (J/N)")
            (read)))
      (B3 (lambda ()
            (print
      "Ist der Betrag groesser gleich 50.00 EUR ? (J/N)")
            (read)))
      (B4 (lambda ()
            (print "Ist der Zweck angegeben ? (J/N)")
            (read)))
      (A1 (lambda () (print "Dem Chef vorlegen")))
      (A2 (lambda ()
            (print
      "Zurueck an die Fachabteilung schicken")))
      (A3 (lambda ()
            (print "Umkontieren auf Konto SONSTIGES")))
      (A4 (lambda ()
            (print "Vorgang an Abteilung 2 abgeben")))
      (A5 (lambda () (print "Buchung vollziehen"))))
;;;;Regeln
  (lambda ()
    (cond((eq? (B1) 'J)
          (cond((eq? (B2) 'J)
                (cond((eq? (B4) 'J) (A5))
                      (#t (cond((eq? (B3) 'J) (A1))
                                (#t (A3) (A5))))))
                (#t (A4))))
        (#t (cond((eq? (B3) 'J) (A2))
                 (#t (cond((eq? (B4) 'J)
                           (A3) (A5))
                          (#t (A1)))))))))
```

Legende:
↪ Tabelle 1.15 S. 83; Bedingungen und Aktionen innerhalb des ET-Konstruktes als lokale Funktionen definiert.

Tabelle 1.17: Programm: `ET-Auszahlung-buchen` — funktional

```
;;;;Begrenzte Eintreffer-ET mit lokalen Konstanten
eval> (define ET-Auszahlung-Buchen (let
     ((B1 '(begin (print
         "Ist die Kontonummer angegeben? (J/N)")
                  (read)))
      (B2 '(begin (print
 "Ist die Kontonummer kleiner gleich 40000 ? (J/N)
                  (read)))
      (B3 '(begin (print
  "Ist der Betrag groesser gleich 50.00 EUR ? (J/N)")
                  (read)))
      (B4 '(begin (print
  "Ist der Zweck angegeben ? (J/N)") (read)))
      (A1 '(print "Dem Chef vorlegen"))
      (A2 '(print "Zurueck an die Fachabteilung schicken"))
      (A3 '(print "Umkontieren auf Konto SONSTIGES"))
      (A4 '(print "Vorgang an Abteilung 2 abgeben"))
      (A5 '(print "Buchung vollziehen")))
;;;;Regeln
    (lambda ()
      (cond((eq? (eval B1) 'J)
               (cond((eq? (eval B2) 'J)
                      (cond((eq? (eval B4) 'J) (eval A5))
                            (#t (cond((eq? (eval B3) 'J)
                                                (eval A1))
                                      (#t (eval A3)
                                          (eval A5))))))
                     (#t (eval A4))))
              (#t (cond((eq? (eval B3) 'J) (eval A2))
                       (#t (cond((eq? (eval B4) 'J)
                                 (eval A3)
                                 (eval A5))
                                (#t (eval A1)))))))))
```

Legende:
↪ Tabelle 1.15 S. 83; Bedingungen und Aktionen innerhalb des ET-Konstruktes als lokale Konstanten definiert.

Tabelle 1.18: Programm: `ET-Auszahlung-buchen` — lokale Konstanten

Notierte Konstrukte

```
eval> (define MURKS-1 lambda (Zahl) (* Zahl 2))
       ==> ERROR ...
 ;Zu viele symbolische Ausdrücke für das
 ; define-Konstrukt, da die Klammern für
 ; das lambda-Konstrukt fehlen.

eval> (define MURKS-2 (lambda (* Zahl 2)))
      ==> ERROR ...
 ;Die Angabe der lambda-Variablen fehlt.
 ; Daher hat das lambda-Konstrukt zu wenige
 ; symbolische  Ausdrücke.

eval> (define (MURKS-3 (* Zahl 2)))
       ==> ERROR ...
 ;Syntaktische Kurzform des define-Kon-
 ; struktes erwartet an der Stelle von
 ; (* Zahl 2) das Symbol Zahl.
 ; Das Symbol Zahl fehlt und eine
 ; schließende Klammer steht an der
 ; falschen Stelle.

eval> (define MURKS-4 (lambda (Zahl)) (* Zahl 2))
        ==> ERROR ...
 ;Zu viele symbolische Ausdrücke für das
 ; define-Konstrukt, da das lambda-Kon-
 ; strukt schon geschlossen wurde bevor
 ; der Funktionskörper notiert ist.

eval> (define MURKS-5 (lambda (Zahl))) (* Zahl 2)
       ==> ERROR ...
 ;Zu wenige symbolische Ausdrücke für das
 ; lambda-Konstrukt. Da das define-Kon-
 ; strukt schon beendet wird, bevor
 ; (* Zahl 2) gelesen wird, fehlt dem
 ; lambda-Konstrukt der Funktionskörper.

eval> (define MURKS-6 (lambda (1) (* Zahl 2)))
        ==> ERROR ...
 ;Ein Zahlatom, hier 1, ist keine
 ; gültige Variable. Zu beachten ist der
 ; Unterschied zwischen einem Symbol
 ; (Literalatom) und einem Zahlatom.
```

Fehler beim Argument

Gewünschtes Konstrukt

```
eval> (define Neuer-Listenkopf
```

```
        (lambda (Liste) (cons 'Konto
                              (cdr Liste)))
eval> (Neuer-Listenkopf '(Huel 12 13))
      ==> (Konto 12 13)
```

Notierte Konstrukte

```
eval> (Neuer-Listenkopf 'Huel 12 13)
      ==> ERROR ...
 ;Die Funktion ist für ein Argument
 ; definiert. Sie wird aber mit drei
 ; Argumenten aufgerufen.

eval> (Neuer-Listenkopf "Huel 12 13)
      ==> Warten!...
 ;Der read-Prozess ist nicht beendet,
 ; da das Ende einer Zeichenkette,
 ; das doppelte Hochkomma, fehlt.

eval> (Neuer-Listenkopf "Huel 12 13")
      ==> ERROR ...
 ;Falscher Typ des Arguments. Das
 ; CDR-Konstrukt ist für eine Zeichen-
 ; kette nicht definiert. Es erwartet
 ; eine Liste bzw. ein Punkt-Paar.

eval> (Neuer-Listenkopf)
      ==> ERROR ...
 ;Kein Argument angegeben, obwohl die
 ; Funktion für ein Argument definiert
 ; wurde. Beachte: Hier ist keine Angabe
 ; ungleich (list)!
 ; eval> (Neuer-Listenkopf (list))
 ;        ==> (KONTO)
```

Fehler beim COND-Konstrukt

Gewünschtes Konstrukt

```
eval> (define Klare-Entscheidung?
        (lambda (Antwort)
          (cond((eq? Antwort 'Nein))
                ((eq? Antwort 'Ja))
                (#t #f))))
eval> (Klare-Entscheidung? 'Ja) ==> #t
eval> (Klare-Entscheidung? 'Enthaltung)
      ==> #f
```

Notiertes Konstrukt

```
eval> (define MURKS-7
        (lambda (Antwort)
          (cond(eq? Antwort 'Nein)
                (eq? Antwort 'Ja)
                (#t #f))))
 ;An der Stelle des Prädikates einer
 ; Klausel steht nicht eine Funktions-
 ; anwendung, also eine Liste, sondern
 ; ein Symbol, hier: eq?. Da das Symbol
 ; eq? als eingebaute Funktion einen Wert
 ; ungleich  #f hat, trifft  stets die
 ; 1. Klausel zu. Der Wert ihres letzten
 ; symbolischen Ausdruckes ist Nein.
 ; Der Wert von MURKS-7 ist daher
 ; stets Nein.
eval> (MURKS-7 'Ja) ==> Nein
eval> (MURKS-7 'Nein) ==> Nein
```

Gewünschtes Konstrukt

```
eval> (define Gewinn-Prognose
        (lambda (sexpr)
          (cond((not (number? sexpr))
                  "Keine Zahl")
                ((number? sexpr)
                   (print
                     "Doppelter Gewinn:")
                   (print (* 2 sexpr))
                   (newline)
                   (print
                     "Dreifacher Gewinn:")
                   (print (* 3 sexpr))
                   (newline)
                   'OK?))))

eval> (Gewinn-Prognose 1000)
      "Doppelter Gewinn:"2000
      "Dreifacher Gewinn:"3000
      OK? ;Wert des letzten Konstruktes
```

Notiertes Konstrukt

```
eval> (define MURKS-8
        (lambda (sexpr)
          (cond((not (number? sexpr))
                  "Keine Zahl")
                ((number? sexpr)
```

```
                    (print
                      "Doppelter Gewinn:")
                    (print (* 2 sexpr))
                    (newline))
                    (print
                      "Dreifacher Gewinn:")
                    (print (* 3 sexpr))
                    (newline)
                    ('OK?))))
 ;Die Klammer nach dem Konstrukt (newline)
 ; schließt die 2. Klausel des cond-Kon-
 ; struktes. Das folgende print-Kon-
 ; strukt ist damit die 3. Klausel mit dem
 ; Symbol print an der Stelle des Prädi-
 ; kates (erste Position). (Un)glücklicher-
 ; weise wird die 3. Klausel nicht er-
 ; reicht, da die beiden vorhergehenden
 ; Klauseln alle möglichen Fälle abfangen.

eval> (MURKS-8 "Alles klar?")
      ==> "Keine Zahl"
eval> (MURKS-8 1000)
      ==> "Doppelter Gewinn:"2000
```

Fehler mit dem quote-Konstrukt

Gewünschtes Konstrukt

```
eval> ((lambda (x y) (list x y))
        'Konto "Schulze AG")
      ==> (Konto "Schulze AG")
```

Notierte Konstrukte

```
eval> ((lambda (x y) (list 'x 'y))
        Konto "Schulze AG")
      ==> ERROR ...
 ;Die Variable Konto ist in der aktuellen
 ; Umgebung nicht definiert. Die Argumente
 ; der anonymen Funktion werden zunächst
 ; evaluiert. Dabei ist für Konto kein Wert
 ; ermittelbar.

eval> ((lambda (x y) (list 'x 'y))
        'Konto "Schulze AG")
      ==> (x y)
 ;Die Argumente des list-Konstruktes sind
 ; aufgrund ihrer Quotierung Konstanten.
 ; Sie entsprechen nicht den Werten an die
```

```
 ; die definierten lambda-Variablen x und y
 ; gebunden werden.

eval> ((lambda ('x 'y) (list x y))
        'Konto "Schulze AG")
      ==> ERROR ...
 ;Es sind keine Symbole als lambda-
 ; Variablen definiert. Durch die Quo-
 ; tierung von x und y besteht die
 ; Schnittstelle des lambda-Konstruktes
 ; aus Konstanten. Diese können keine
 ; Symbol-Wert-Assoziationen abbilden.
```

1.2.5 Zusammenfassung: `let`, `do` und `map`

Wir definieren ein eigenes Konstrukt (Programm) indem wir eingebaute LISP-Konstrukte und/oder selbstdefinierte Konstrukte mit einander kombinieren. Ob das Konstrukt „zweckmäßig“ definiert ist, zeigt sich nicht nur in Bezug auf die spezifizierte Aufgabe, die es korrekt abzubilden gilt, sondern auch daran, ob das Konstrukt später einfach änderbar und ergänzbar ist. Der gesamte Software-Lebenszyklus ist Bewertungsmaßstab.

Für den symbolischen Ausdruck Liste verfügt ein LISP-System über den eingebauten Konstruktor `cons` mit den zugehörenden Selektoren `car` und `cdr` sowie über ein Prädikat, das feststellt, ob die leere Liste vorliegt. Das Prädikat `eq?` prüft die Identität zweier symbolischer Ausdrücke. Für den Test auf Gleichheit von zwei Listen gibt es das Prädikat `equal?`.

Konstrukte sind verknüpfbar als *Sequenz*, *Selektion* und *Iteration*.

Zur Abbildung der Sequenz dienen die Komposition von Funktionen und Konstrukte wie `begin` oder `begin0`. Das `lambda`-Konstrukt hat in seinem Teil Rechenvorschrift eine implizit definierte Sequenz, da es dort eine Folge symbolischer Ausdrücke der Reihe nach auswertet. Das `let`-Konstrukt als syntaktisch vereinfachte Anwendung einer anonymen Funktion (Applikation eines `lambda`-Konstruktes) bindet seine lokalen Variablen nicht in einer Sequenz. Erst das `let*`-Konstrukt vollzieht diese Bindung sequentiell.

Die Selektion ist mit Konstrukten wie `if`, `cond` oder `case` realisierbar.

Die Iteration ist einerseits mit dem umfassenden `do`-Konstrukt und andererseits mit `map`-Konstrukten (*mapping functions*) abbildbar. Die Iteration ist als Sonderfall der Rekursion betrachtbar. Sie wird daher auch als rekursive Formulierung notiert.

Charakteristische Beispiele für Abschnitt 1.2

```
eval> (cond(1 2 3) (4 5 6)) ==> 3
eval> (caddr '(lambda (x y) (- y x)))
      ==> (- y x)
eval> (cons 23.45 67.89)
      ==> (23.45 . 67.89)
eval> (eq? (cons 'Meyer-AG '(Mueller-OHG))
                 '(Meyer-AG Mueller-OHG))
      ==> #f
eval> (equal? (cons 'Meyer-AG '(Mueller-OHG))
                    '(Meyer-AG Mueller-OHG))
      ==> #t
eval> (define 1+ 7)
eval> (+ 1+ 12) ==> 19
eval> (define *3
        (lambda (wert)
          (let ((dreifach (lambda (x) (* 3 x))))
            (dreifach wert))))
eval> (*3 5) ==> 15
eval> (let* ((x 2) (y (+ x 1))) (* x y))
      ==> 6
eval> (and (not (and (not 3))))
      ==> #t
eval> (case 'M ((Mein Dein Sein) #t)
               (else "Nicht dabei!"))
      ==> "Nicht dabei!"

eval> (do ((i 2 (- i 1)))
          ((begin (print "Ja")
                  (newline)
                  (= 0 i))))
      ==>
        "Ja"
        "Ja"
        "Ja"
eval> (map list '(A B C D))
      ==> ((A) (B) (C) (D))

eval> (for-each print
        '("Gestern+" "Heute+" "Morgen!"))
      ==> "Gestern+""Heute+""Morgen!"
```

1.3 Rekursion als Problemlösungsmethode

Nimmt eine Definition auf sich selbst Bezug, dann liegt direkte Rekursion vor. Ist der Selbstbezug über eine Aufruffolge von Definitionen gegeben, dann spricht man von indirekter oder gegenseitiger Rekursion.

```
eval> (define Laenge
        (lambda (Liste)
          (cond((null? Liste) 0)
                (#t (+ 1
                  (Laenge (cdr Liste)))))))

eval> (Laenge '(Meyer-AG Mueller-OHG Schulze-GmbH))
      ==> 3
eval> (Laenge '((A) (B (C (D)))))
      ==> 2
```

Legende:
Bestimmung der Länge einer Liste, d. h. Ermittlung der Anzahl ihrer Elemente.

Tabelle 1.19: Programm: Beispiel einer rekursiven Definition

Problemlösungen über rekursive Definitionen sind für LISP-Programme kennzeichnend. Schon bei der Definition von Syntax und Semantik zeigte sich die Einfachheit und Kürze rekursiver Definitionen.

Eines der oft erwähnten Beispiele für die Erläuterung der Rekursion ist die Definition der Funktion `Laenge` (↪ z. B. [180], S. 27ff oder [135], S. 63ff). Sie bestimmt die Länge einer Liste, d. h. die Anzahl der Elemente.

Das `null?`-Prädikat im `Laenge`-Konstrukt (↪ Programm in Tabelle 1.19 S. 114) stellt fest, ob eine leere Liste vorliegt. Es entspricht folgender Definition:

```
eval> (define null? (lambda (x) (eq? x (list))))
```

Die rekursive Methode ist immer dann nützlich, wenn eine Familie „verwandter" Probleme vorliegt, von denen eines so einfach ist, dass sich die Lösung direkt angeben lässt. Diesen trivialen Fall bezeichnet *Douglas R. Hofstadter* treffend mit „Embryonal-Fall" (↪ [92]), da sich die Lösung des Gesamtproblems aus ihm „entwickelt".

So lässt sich die Länge der leeren Liste sofort mit 0 angeben. Die Länge einer Liste mit z. B. 3 Elementen ist zurückführbar auf das einfachere Problem der Ermittlung der Länge einer Liste mit 2 Elementen, da die 3-elementige Liste um 1 Element länger ist.

Generell stellen sich die beiden Kernprobleme für das Auffinden einer *Rekursions-Route*:

1. Definition von trivialen Lösungen und
2. Definition der Verknüpfung von Lösungsschritten.

1. Definition von trivialen Lösungen
Welches ist der einfache Fall (bzw. welches sind die einfachen Fälle), bei dem (denen) die Lösung(en) direkt angebbar ist (sind)?

- Woran erkennt man einen einfachen Fall?
- Wie lautet die zugehörende Lösung?

2. Definition der Verknüpfung von Lösungsschritten
Welches ist die Verknüpfung zwischen einem typischen und dem nächst einfacheren Fall?

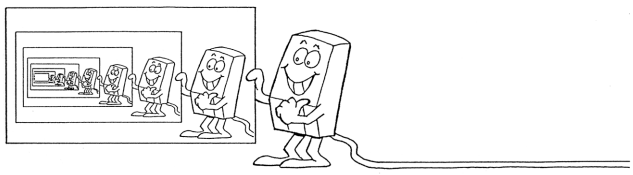
- Wie kommt man von einem typischen Fall genau einen Schritt näher zu einem einfachen Fall?
- Wie konstruiert man die Lösung für den vorliegenden Fall aus der „unterstellten" Lösung für den einfacheren Fall?

Beim Fall `Laenge` lauten die Antworten auf diese Fragen:

- Der einfache Fall ist gegeben, wenn die Liste leer ist, d. h. das `null?`-Konstrukt liefert den Wert `#t`.
- Bei der leeren Liste ist die Länge gleich 0.
- Verkürze die gegebene Liste um ein Element.
- Addiere 1 zur Lösung der um ein Element verkürzten Liste.

Entscheidend ist, dass die Rekursions-Route tatsächlich zu trivialen Fällen führt. Anhand eines rekursiven Akronyms soll der umgekehrte, in der Regel der ungewollte Weg aufgezeigt werden. Man nehme zum Spaß an, lambda stehe für folgendes rekursive Akronym:
lambda ≡ *lambda auch mein bestens definierbares Arbeitsmittel*
Ersetzt man das *lambda* der rechten Seite durch die Definition, dann erhält man nach zweimaliger Anwendung folgendes Ungetüm:

(((lambda auch mein bestens definierbares Arbeitsmittel) auch mein bestens definerbares Arbeitsmittel) auch mein bestens definierbares Arbeitsmittel).



Bei der Anwendung einer rekursiven Definition sind zunächst Zwischenlösungen für die Teilprobleme zu vermerken, bis eine triviale Lösung erreicht wird. Von dieser können dann „rückwärts“ die vermerkten Teilprobleme gelöst werden. Im allgemeinen Rekursionsfall wächst daher der Speicherbedarf mit der Zahl der erforderlichen Zwischenschritte. Die Anzahl der geschachtelten Aufrufe wird als Rekursionstiefe der Funktion bezeichnet. Im speziellen Fall der sogenannten Restrekursion (engl.: *tail-recursion*) ist der Speicherbedarf konstant. Er ist unabhängig vom Wert des Argumentes und damit unabhängig von der Rekursionstiefe. Bei der Restrekursion ist das Teilproblem-Ergebnis der größten Rekursionstiefe das Problem-Gesamtergebnis (↪ Hinweis Restrekursion S. 116). Effiziente LISP-Systeme erkennen, ob eine Definition eine Restrekursion darstellt, und arbeiten diese dann mit konstantem Speicherbedarf entsprechend einer Iteration ab. Für Scheme-Systeme wird diese Eigenschaft gefordert (↪ [151]). Der Benutzer kann sich somit auf rekursive Formulierungen konzentrieren, ohne ein unnötiges Wachsen des Speicherbedarfs im Restrekursions-Fall in Kauf nehmen zu müssen. Ein Beispiel ist die Funktion `Saegezahn` (↪ Programm in Tabelle 1.20 S. 118).

Hinweis: Restrekursion
Das obige `Laenge` (↪ Programm in Tabelle 1.19 S. 114) ist nicht restrekursiv definiert. Mit Hilfe eines Akkumulators ist eine restrekursive Lösung leicht definierbar, wie das folgende Beispiel zeigt:

```
eval> (define rrLaenge
        (lambda (liste) (Zaehlung liste 0)))
eval> (define Zaehlung
        (lambda (l Akkumulator)
          (cond((null? l) Akkumulator)
                (#t (Zaehlung
                      (cdr l)
                      (+ 1 Akkumulator))))))
eval> (rrLaenge '(A B C D)) ==> 4
```

Hinweis: *PLT-Scheme Debugger*
Zum Verständnis einer rekursiven Definition ist es hilfreich, die Aufruffolgen mit ihren Argumenten und dem zugehörenden Rückgabewert darzustellen. *PLT-Scheme* verfügt über einen leistungsfähigen *Debugger*[14], der die Aufruffolgen in einzelnen Schritten (*Steps*) anzeigt. Für das definierte `rrLaenge`-Konstrukt zeigt Abbildung A.3 S. 470 einen Zwischenschritt.

Hinweis: *PC-Scheme* `TRACE`-Konstrukt
Das `TRACE`-Konstrukt in *PC-Scheme* ermöglicht eine Ausgabe der Aufruffolgen. Dabei protokolliert `TRACE-ENTRY` jeden Aufruf und `TRACE-EXIT` jedes Verlassen der Funktion. `TRACE-BOTH` dokumentiert beides; den Aufruf und das Verlassen der Funktion.

[14] Ein *Debugger* (engl.: *bug* im Sinne von Fehler) ist ein Diagnose-Tool.

Bezogen auf das `Laenge`-Konstrukt (↪ Programm in Tabelle 1.19 S. 114) erhält man folgende Unterstützung:

```
eval> (TRACE-BOTH Laenge) ==> OK
eval> (Laenge '(Meyer-AG Mueller-OHG Schulze-GmbH))
        ==>
 Argument 1: (Meyer-AG Mueller-OHG Schulze-GmbH)
>>> Entering #<procedure Laenge>
 Argument 1: (Mueller-OHG Schulze-GmbH)
>>> Entering #<procedure Laenge>
 Argument 1: (Schulze-GmbH)
>>> Entering #<procedure Laenge>
 Argument 1: ()
<<< Leaving #<procedure Laenge> with value 0
 Argument 1: ()
<<< Leaving #<procedure Laenge> with value 1
 Argument 1: (SCHULZE-GMBH)
<<< Leaving #<procedure Laenge> with value 2
 Argument 1: (Mueller-OHG Schulze-GmbH)
<<< Leaving #<procedure Laenge> with value 3
 Argument 1: (Meyer-AG Mueller-OHG Schulze-GmbH)
3
```

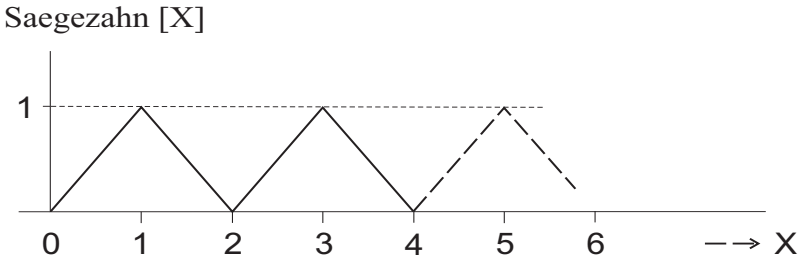
Rekursive Definitionen sind in der Mathematik gebräuchlich. Man notiert z. B. eine Sägezahn-Kurve wie folgt:[15]

$$Saegezahn[x] = \begin{cases} 0 & falls \quad x < 0 \\ x & falls \quad 0 \leq x < 1 \\ 2-x & falls \quad 1 \leq x < 2 \\ Saegezahn[x-2] & falls \quad 2 \leq x \end{cases}$$

Die Abbildung 1.22 S. 118 zeigt die Kurve der obigen restrekursiv definierten Sägezahnfunktion (↪ mathemathische Notation S. 117). Die in der Mathematik gebräuchliche Notation ist direkt in LISP übernehmbar, wobei die sequentielle Abarbeitung des `cond`-Konstruktes einen Verzicht auf die kleiner-gleich-Schranken erlaubt (↪ Programm in Tabelle 1.20 S. 118).

Die Rekursivität ist eine Eigenschaft der Lösung eines Problems. Sie ist nicht eine Eigenschaft des Problems selbst. Wir vertiefen die rekursive Problemlösungs-Methode anhand von Beispielen. Als erstes Beispiel ist eine Funktion definiert, die in einem beliebigen Text ermittelt, wie oft ein vorgegebenes Wort vorkommt. Dabei erörtern wir die Technik der rekursiven Funktionsanwendung auf die Restliste eines Listen-Argumentes (↪ Abschnitt 1.3.1 S. 119). Das zweite Beispiel ist eine Funktion, die in

[15] Hinweis: Es handelt sich um eine mathematische Notation und nicht um eine LISP-Notation. Um den Unterschied zu verdeutlichen, wurden in der mathematischen Notation eckige statt runden Klammern verwendet.



Legende:
Grafische Darstellung der restrekursiv definierten Funktion `Saegezahn`
↪ mathemathische Notation S. 117.

Abbildung 1.22: Beispiel: `Saegezahn`-Funktion

```
eval> (define Saegezahn
        (lambda (x)
          (cond((< x 0) 0)
               ((< x 1) x)
               ((< x 2) (- 2 x))
               (#t (Saegezahn(- x 2))))))

eval> (Saegezahn 4.5) ==> 0.5
```

Legende:
↪ Abbildung 1.22 S. 118.

Tabelle 1.20: Programm: Restrekursiv definierte Funktion `Saegezahn`

einem Text ein vorgegebenes Wort durch ein neues ersetzt, wobei der Text als Liste mit Sublisten abgebildet ist. Es geht dabei um das Absuchen einer Baumstruktur (↪ Abschnitt 1.3.2 S. 122). Im dritten Beispiel befassen wir uns mit der Formulierung einer rekursiven lokalen Funktion. Dazu „navigieren“ wir entlang eines Binärbaums (↪ Abschnitt 1.3.3 S. 124). Im vierten Beispiel schachteln wir rekursive Aufrufe. Als Beispiel dient die Definition der Funktion `Umdrehen`. Sie entspricht der eingebauten Funktion `reverse` (↪ Abschnitt 1.3.4 S. 128).

1.3.1 Funktionsanwendung auf die Restliste

Beispiel: Wortzählung in einem Text mit einfacher Listenstruktur

Es ist eine Funktion `Wortzaehlung` zu definieren, die feststellt, wie oft ein vorgegebenes Wort in einem beliebigen Text vorkommt, der als Liste vorliegt. Als Textbeispiel dient §7 Abs. 1 BGB (Bürgerliches Gesetzbuch).[16]

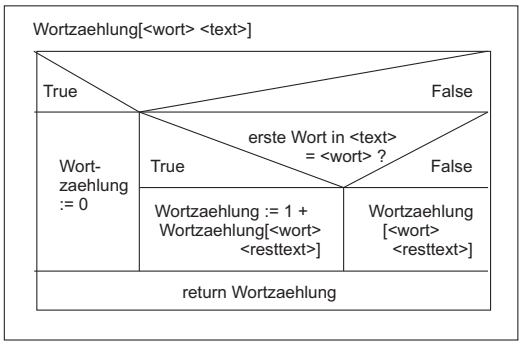
```
eval> (define BGB-7.1 '(Wer sich an einem Orte
        ständig niederläßt begründet an diesem
        Orte seinen Wohnsitz))
eval> (Wortzaehlung 'Orte BGB-7.1) ==> 2
eval> (Wortzaehlung 'Wohnsitz BGB-7.1) ==> 1
```

Entsprechend der rekursiven Problemlösungs-Methode suchen wir zunächst triviale Fälle und ihre Lösungen. Danach ermitteln wir eine Konstruktionsvorschrift, die das Problem in Richtung auf triviale Fälle vereinfacht (Bestimmung der Rekursions-Route).

Der triviale Fall ist gegeben, wenn der Text kein Wort enthält. Dann ist die Lösung der Wert 0. Enthält der Text Wörter, dann soll zunächst das erste Wort des Textes mit dem vorgegebenen Wort verglichen werden. Liegt Übereinstimmung vor, dann ist die gesuchte Lösung um 1 größer als das Ergebnis der Wortzählung im Resttext. (Die bisherige Definition entspricht dem erläuterten `Laenge`-Konstrukt). Ist keine Übereinstimmung gegeben, dann ist die gesuchte Lösung gleich dem Ergebnis der Wortzählung im Resttext. Der Resttext ist die Restliste, d. h. der Text ohne das erste Wort. Die Abbildung stellt diese Lösung als Struktogramm dar. Die Beschriftung orientiert sich dabei an der von *I. Nassi* und *B. Shneiderman* für ihr Fakultätsbeispiel gewählten Notation (↪ [138]). Diese Notation lässt allerdings die wesentliche Unterscheidung zwischen der Applikation einer Funktion und der Darstellung des Funktionswertes nur schwer erkennen, wie auch die Legende in Abbildung 1.23 S. 120 zeigt.

Entsprechend dem Struktogramm ist die Funktion, wie folgt, zu definieren:

[16] Schreibweise mit ß entspricht dem Originaltext.



Legende:
Grafische Darstellung einer Rekursion als Struktogramm DIN 66261 (Struktogramme ↪ [138])
`Wortzaehlung := 0` ≡ Der Wert der Funktion ist 0.
`Wortzaehlung[<wort> <text>]` ≡ Die Funktion wird angewendet auf die Argumente `<wort>` und `<text>`.
`return Wortzaehlung` ≡ Der Funktionswert wird ausgegeben.

Abbildung 1.23: Struktogramm `Wortzaehlung`

```
eval> (define Wortzaehlung
        (lambda (Wort Text)
          (cond ((null? Text) 0)
                ((eq? Wort (car Text))
                 (+ 1 (Wortzaehlung Wort
                                    (cdr Text))))
                (#t (Wortzaehlung Wort
                                  (cdr Text))))))
eval> (define BGB-7.1 '(Wer sich an einem Orte
        ständig niederläßt begründet an diesem
        Orte seinen Wohnsitz))
eval> (Wortzaehlung 'Orte BGB-7.1) ==> 2
```

Die Lösung kann auch durch eine Entscheidungstabelle (ET) mit einer Aktion „`Wiederhole ET`“ dargestellt werden (↪ Tabelle 1.21 S. 121). Eine solche Notation verdeutlicht jedoch nicht die rekursive Definition. Diese zeigt Tabelle 1.22 S. 121. Hier ist allerdings wie beim Struktogramm das Problem, die Applikation der Funktion vom Rückgabewert der Funktion deutlich zu unterscheiden. Dass die Iteration als Sonderfall von selbstbezüglichen Funktionen betrachtbar ist, verdeutlicht die Rekursion mit der Funktionsanwendung auf die Restliste. Wir können mit dieser Notation die Konstrukte für das klassische *Mapping* (↪ S. 100) definieren.

```
eval> (define for-each (lambda (funktion liste)
        (cond((null? liste) (void))
```

Akkumulator := 0

	ET-Wortzaehlung	R1	R2	R3
B1	Enthält `<text>` kein Wort?	J	N	N
B2	Erste Wort im `<text> = <wort>`?	–	J	N
A1	Akkumulator := 1 + Akkumulator		X	
A2	`<text> := <text reduziert um erstes wort>`		X	X
A3	Wiederhole ET-Wortzaehlung		X	X
A4	Gebe Akkumulator aus	X		

Legende:
ET-Symbole ↪ Abschnitt 13 S. 80
Formale Vollständigkeitsüberprüfung:
2 einwertige Regeln (R2 und R3) ↪ 2 Fälle
1 zweiwertige Regeln (R1) ↪ 2 Fälle
∑ 4 Fälle
≡ 4 Regeln

Tabelle 1.21: Iteration dargestellt in ET-Technik — Funktion `Wortzaehlung`

	ET-Wortzaehlung `[<wort> <text>]`	R1	R2	R3
B1	Enthält `<text>` kein Wort?	J	N	N
B2	Erste Wort im `<text> = <wort>`?	–	J	N
A1	Wortzaehlung := 0	X		
A2	Wortzaehlung := 1 + `call` ET-Wortzaehlung `[<wort> <text reduziert um erstes wort>]`		X	
A3	`call` ET-Wortzaehlung `[<wort <text reduziert um erstes wort>]`			X
A4	Return Wortzaehlung	X	X	X

Legende:
`call` ≡ Verweist auf die Funktionsapplikation
ET-Symbole ↪ Abschnitt 13 S. 80
Formale Vollständigkeitsüberprüfung:
2 einwertige Regeln (R2 und R3) ↪ 2 Fälle
1 zweiwertige Regeln (R1) ↪ 2 Fälle
∑ 4 Fälle
≡ 4 Regeln

Tabelle 1.22: Rekursion dargestellt in ET-Technik — Funktion `Wortzaehlung`

```
              (#t (funktion (car liste))
                  (for-each funktion (cdr liste))))))

eval> (for-each (lambda(x) (print x) (newline))
        '(1 2 3)) ==>
      1
      2
      3
```

Das `map`-Konstrukt sammelt die Ergebnisse der Funktionsanwendung und gibt sie als Liste zurück. Dieses Sammeln bilden wir mit dem `cons`-Konstrukt ab.

```
eval> (define map (lambda (funktion liste)
        (cond((null? liste) (list))
              (#t (cons (funktion (car liste))
                        (map funktion
                             (cdr liste)))))))
eval> (map (lambda (x)
             (cons x (cons x (list))))
           '(A B C)) ==> ((A A) (B B) (C C))
```

Die rekursive Definition des `map`-Konstruktes zeigt eine häufig vorkommende Lösungsstruktur. Salopp „LISP-isch“ formuliert: *CDRing down the list and CONSing up the answer*.

1.3.2 Verknüpfung von Teilproblem-Lösungen

Beispiel: Änderung eines Textes, der als Baumstruktur vorliegt

Wir definieren eine Funktion `Ersetzung`, die in einem Text ein bestimmtes Wort durch ein neues Wort ersetzt. Der Text ist als Baum strukturiert, d. h. als eine Liste, die wiederum Listen als Elemente enthält. Als Textbeispiel dient wie in Abschnit 1.3.1 S. 119 wieder §7 BGB.

```
eval> (define BGB-7
        '(((1) Wer sich an einem Orte
               ständig niederläßt
               begründet an diesem Orte
               seinen Wohnplatz)
          ((2) Der Wohnplatz kann
               gleichzeitig an mehreren
               Orten bestehen)
          ((3) Der Wohnplatz wird aufgehoben
               wenn die Niederlassung
               mit dem Willen aufgehoben wird
               sie aufzugeben)))
```

In diesem Text ist das Wort `Wohnplatz` falsch; es ist durch `Wohnsitz` zu ersetzen. Für das zu definierende Konstrukt wird offensichtlich

eine Schnittstelle mit drei Parametern benötigt, die wir, wie folgt, benennen:

`Objekt` ≡ das zu ersetzende alte Wort
`Ersatz` ≡ das neue Wort
`Struktur` ≡ der strukturierte Text, in dem ersetzt wird

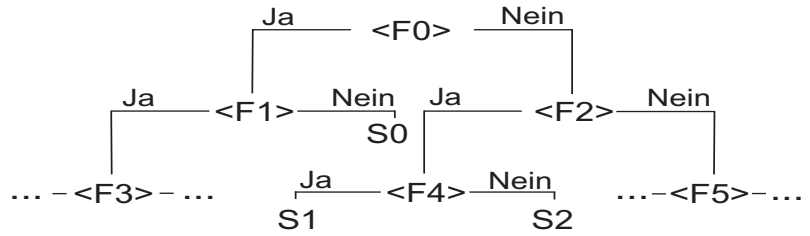
Entsprechend der rekursiven Problemlösungs-Methode suchen wir zunächst nach trivialen Fällen, bei denen die Lösung bekannt und damit direkt angebbar ist. Wenn die `Struktur` nur aus dem zu ersetzenden Wort `Objekt` besteht, dann ist die Lösung bekannt und wir brauchen nur `Ersatz` zurückzugeben. Ist die `Struktur` keine Liste, sondern ein elementares Element z.B. eine Zahl oder ein Symbol und ist dieses Element nicht gleich dem `Objekt`, dann ist die `Struktur` selbst die Lösung. Anders formuliert: Ist die `Struktur` ein Atom (↪ Tabelle 1.9 S. 67) und trifft nicht der erste triviale Fall zu, dann ist die `Struktur` der Rückgabewert. Ob ein Atom vorliegt betrachten wir als Frage, ob keine Punkt-Paar-Struktur (↪ Abbildung 1.15 S. 60) vorliegt und zwar mit dem Konstrukt `pair?`.

Betrachten wir das erste Element von `Struktur`, dann kann es ein Atom oder wieder eine Liste sein. Für den Fall eines Atomes haben wir die Lösung mit der Definition der trivialen Fälle abgedeckt. Für eine Liste ist der Ersetzungsvorgang sowohl für ihr erstes Element als auch für ihre Restliste zu vollziehen. Das Ergebnis ist die Verknüpfung der Ersetzung im ersten Element mit der Ersetzung in der Restliste. Diese Verknüpfung ist mit dem Konstruktor `cons` abbildbar. Wir können die Ersetzung in einer komplexen `Struktur` auf zwei einfachere Probleme zurückführen:

1. auf die Ersetzung im ersten Element von `Struktur` und
2. auf die Ersetzung im Rest von `Struktur`.

Die Gesamtlösung ist die Verknüpfung der beiden Teillösungen. Wir definieren daher:

```
eval> (define Ersetzung
        (lambda (Objekt Ersatz Struktur)
          (cond
            ((eq? Objekt Struktur)
                              Ersatz)
            ((not (pair? Struktur))
                              Struktur)
            (#t (cons
                 (Ersetzung
                  Objekt Ersatz (car Struktur))
                 (Ersetzung
                  Objekt Ersatz (cdr Struktur))))
            )))
eval> (Ersetzung 'Wohnplatz 'Wohnsitz BGB-7)
```



Legende:
F0, F1, F2, ... ≡ Fragen
S0, S1, S2, ... ≡ Systemantworten

Abbildung 1.24: Skizze eines Binärbaumes

```
 ==>
(((1) Wer sich an einem ...  Wohnsitz)
 ((2) Der Wohnsitz ... bestehen)
 ((3) Der Wohnsitz ... aufzugeben))
```

1.3.3 Rekursive Konstrukte für rekursive Strukturen

Die Rekursion ist die „geborene“ Lösungsmethode für die Handhabung von Baumstrukturen. Im Allgemeinen sind Operationen, die sich auf einen Baum beziehen, wiederum auf dessen Teilbäume anzuwenden und zwar so lange, bis die Baumblätter erreicht werden. Anhand eines simplen Beratungssystems wird deutlich, dass die rekursive Lösungsmethode zweckmäßig für Baumstrukturen ist.

Beispiel: Ja-Nein-Navigation entlang einer Baumstruktur

Zu definieren sei ein Beratungssystem, das auf seine Fragen als Dialogeingabe „Ja“ oder „Nein“ erwartet. Die Fragen sind hierarchisch strukturiert (↪ Abbildung 1.24 S. 124).

Das Beratungssystem basiert auf der rekursiven Struktur: `(<frage>(<ja-baum>)(<nein-baum>))` Der `<ja-baum>` und der `<nein-baum>` haben beide wieder diese Struktur. Beim Abarbeiten des Baumes wird eine Systemantwort erkannt, wenn der Baum nur noch aus einem Element besteht; d. h. `(length <baum> )` ist gleich 1. [Hinweis: `length` ist eine eingebaute LISP-Funktion. Sie entspricht dem Konstrukt `Laenge` (↪ Tabelle 1.19 S. 114)] Das Durchlaufen dieses Binärbaums steuert die rekursive Funktion `Navigation`. Nach der rekursiven Problemlösungsmethode konstruieren wir die Lösung auf der Basis trivialer Fälle. Hier ist der triviale Fall gegeben, wenn der Binärbaum so weit abgearbeitet ist, dass ein Baumblatt erreicht ist, d.h. eine Systemantwort auszugeben ist. Wir definieren daher ein Prädikat, das feststellt, ob ein Baumblatt gegeben ist.

```
eval> (define Baumblatt?
        (lambda (Baum)
          (eq? 1 (length baum))))
```

Abhängig von der Dialogantwort ist entweder im `<ja-baum>` oder im `<nein-baum>` das „Navigieren“ fortzusetzen. Wäre der `<ja-baum>` weiter zu verfolgen, dann ist ein Selektor erforderlich, der aus dem ursprünglichen Baum den `<ja-baum>` ermittelt. Bezogen auf die Liste, die den Baum abbildet, ist das zweite Element zu selektieren. Für den `<nein-baum>` ist ein analoger Selektor erforderlich. Wir definieren eine Funktion `Verzweigung`, die abhängig von der Dialogantwort die entsprechende Selektorfunktion ermittelt. Der Wert der Funktion `Verzweigung` ist selbst eine Funktion, nämlich die jeweilige Selektorfunktion. `Verzweigung` ist daher eine Funktion höherer Ordnung.

```
eval> (define Verzweigung
                (lambda ()
                  (cond((Zustimmung?)
                        Selektor-Ja-Baum)
                       (#t Selektor-Nein-Baum))))
```

Die Funktion `Zustimmung?` ist mit dem `case`-Konstrukt (↪ S. 88) definiert. Entsprechend der gewählten Datenstruktur für die Abbildung des Binärbaums sind die Selektorfunktionen, wie folgt, zu definieren:

```
eval> (define Selektor-Ja-Baum
                (lambda (Baum)
                  (car (cdr Baum))))
eval> (define Selektor-Nein-Baum
                (lambda (Baum)
                  (car (cdr (cdr Baum)))))
```

Mit `Verzweigung` als Funktion höherer Ordnung ist das Konstrukt `Navigation`, wie folgt, formulierbar:

```
eval> (define Navigation
        (lambda (Baum)
          (cond((Baumblatt? Baum)
                (System-Antwort Baum))
               (#t (System-Frage Baum)
                   (Navigation
                     ((Verzweigung) Baum))))))
```

Die Konstrukte `System-Frage` und `System-Antwort` definieren wir mit den entsprechenden Selektoren, wie folgt:

```
eval> (define Selektor-Frage
        (lambda (Baum) (car Baum)))
eval> (define System-Frage
        (lambda (Baum)
          (print (Selektor-Frage Baum))))
```

```
eval> (define Selektor-Baumblatt
        (lambda (Baum) (car Baum)))
eval> (define System-Antwort
        (lambda (Baum)
          (print "Systemantwort: "
            (Selektor-Baumblatt Baum))))
```

In der Definition von `Navigation` führt die Applikation von `(Verzweigung)` zur Rückgabe der anzuwendenden Selektorfunktion, so dass `((Verzweigung) Baum)` den zutreffenden „Restbaum“ ermittelt. Wir wenden auf diesen Restbaum die Funktion `Navigation` erneut an. Diese Vorgehensweise entspricht der schon erörterten Technik der Funktionsanwendung auf die Restliste (↪ Abschnitt 1.3.1 S. 119). Der Unterschied besteht in der Ermittlung des „Restes“; d. h. des einfacheren Problems im Sinne der Rekursionsroute.

Der skizzierte Lösungsansatz führt zu einer Anzahl von Funktionen, die im Zusammenhang mit dem `Beratungssystem`-Konstrukt zu definieren sind. Außerhalb dieses Konstruktes sind diese Funktionen nicht erforderlich. Mit Hilfe einer `let`-Formulierung (↪ Abschnitt 1.2.2 S. 72) können die Funktionen wie `Selektor-Ja-Baum`, `Selektor-Nein-Baum`, `Baumblatt`, `Verzweigung`, `Navigation` etc. als lokale Variablen definiert werden. Wir vermeiden damit, dass diese Konstrukte auf der LISP-System-Ebene (*top level*) bekannt sind und z. B. versehentlich durch namensgleiche Symbole überschrieben werden. Mit der `let`-Notation sind ihre Wert-Assoziationen nicht in der Startumgebung eingetragen, sondern in der Umgebung, die durch die `let`-Konstruktion aufgebaut wird. Da das Konstrukt `Navigation` rekursiv ist, wird das `letrec`-Konstrukt benötigt.

Das Programm in Tabelle 1.23 S. 127 zeigt die Lösung mit dem `letrec`-Konstrukt. Sie weist darüber hinaus einige kleine Verbesserungen auf, z. B. steht statt der Schachtelung von `(car (cdr (cdr Baum)))` der Selektor `(list-ref Baum 2)`.

Hinweis: `list-ref`-Konstrukt

In Scheme ermittelt das `list-ref`-Konstrukt das n-te Element einer Liste, wobei $n = 0$ das erste Element der Liste selektiert, d. h. dem `car`-Konstrukt entspricht. Die Positionszählung der Elemente beginnt mit Null. Das `list-ref`-Konstrukt ist *zero-based*.

```
eval> (list-ref '(a b c) 2) ==> c
eval> (list-ref '(a b c) 0) ==> a
```

Häufig hat dieser Selektor den Namen `nth` (*nth element of list*). Ihn gibt es allerdings mit unterschiedlichen Schnittstellen. In *Common LISP* (↪ [174]) hat er die Parameterfolge: `(nth <positionsnummer> <liste>)`. In *TLC-LISP* gilt die umgekehrte Reihenfolge: `(nth <liste> <positionsnummer>)`. Darüber hinaus kommt er in der Ausprägung *zero based* oder *one-based* vor.

```
;;;;Beratungssystem konstruiert mit dem Binärbaum:
;;;; (<frage> <ja-baum> <nein-baum>)
eval> (define Beratungssystem (lambda (Baum)
       (letrec
         ((Selektor-Ja-Baum
            (lambda (x) (list-ref x 1)))
          (Selektor-Nein-Baum
            (lambda (x) (list-ref x 2)))
          (Selektor-Frage (lambda (x) (car x)))
          (Selektor-Baumblatt
            (lambda (x) (car x)))
          (Baumblatt?
            (lambda (x) (eq? 1 (length x))))
          (Zustimmung? (lambda ()
            (case (read)
              ((J JA Y YES T) #t) (else #f))))
          (Verzweigung (lambda ()
            (cond((Zustimmung?) Selektor-Ja-Baum)
                 (#t Selektor-Nein-Baum))))
          (System-Frage (lambda (x)
            (print (Selektor-Frage x))))
          (System-Antwort (lambda (x)
            (print "Systemantwort: ") (newline)
            (Selektor-Baumblatt x)))
          (Navigation (lambda (x)
            (cond((Baumblatt? x)
                    (System-Antwort x))
                 (#t (System-Frage x)
                      (Navigation
                        ((Verzweigung) x)))))))
         (Navigation Baum))))
```

Tabelle 1.23: Programm: Beratungssystem konstruiert mit einem Binärbaum

Das Programm in Tabelle 1.23 S. 127 wenden wir im Folgenden für eine skizzenhafte Beratung zum Erwerb eines Fahrrades an:

```
eval> (define Fahrrad-Analyse
        '("Hat es einen Diamantrahmen?"
            ("Ist der Rahmen hoeher als 56 cm?"
              ("Ungeeignet fuer ein Kind!")
              ("Notfalls fuer ein Kind verwendbar!"))
   ("Bedingt geeignet - nur fuer kleine Personen!")))

eval> (Beratungssystem Fahrrad-Analyse) ==>
      "Hat es einen Diamantrahmen?"Nein
      "Systemantwort: "
      "Bedingt geeignet - nur fuer kleine Personen!"
```

1.3.4 Geschachtelte Rekursion

Beispiel: Umdrehen der Reihenfolge der Elemente in einer Liste

Die Funktion `reverse` hat als Wert eine Liste, die die Elemente ihres Argumentes in umgekehrter Reihenfolge aufweist.

```
eval> (reverse '(A (B C) D)) ==> (D (B C) A)
```

Wir definieren diese Funktion selbst und zwar rekursiv. Um die eingebaute Funktion nicht zu überschreiben, nennen wir unsere Funktion `Umdrehen`.

Im Sinne der rekursiven Problemsicht halten wir zunächst fest, dass das Konstrukt `Umdrehen` bei einer Liste ohne Elemente den Wert `()` hat. Dann unterstellen wir, dass für eine Liste eine Teillösung vorliegt, nämlich ihre umgedrehte Restliste. Mit dieser Teillösung ist dann die gesamte Problemlösung bekannt. Sie ist gleich das erste Element der Liste, an das Ende der unterstellten Teillösung angehängt. Benötigt wird eine Funktion, die ein solches Anhängen an eine Liste ermöglicht. Das entsprechende Konstrukt heißt `append` (deutsch: hinzufügen). Es verknüpft zwei Listen miteinander.

```
eval> (append '(1 2 3) '(4 5)) ==> (1 2 3 4 5)
```

Damit ist die Definition von `Umdrehen`, wie folgt, formulierbar:

```
eval> (define Umdrehen
        (lambda (x)
          (cond ((null? x) (list))
                (#t (append
                     (Umdrehen (cdr x))
                     (list (car x)))))))
eval> (Umdrehen '(1 2 3)) ==> (3 2 1)
```

Das `append`-Konstrukt ist ebenfalls rekursiv definierbar. Wir nennen es (um ein Überschreiben zu vermeiden) `Anhaengen`. Ein trivialer Fall

ist gegeben, wenn der Wert eines Argumentes von Anhaengen die leere Liste ist. Die zugehörige Lösung ist dann die jeweils andere Liste. Da mit dem cons-Konstrukt ein Mittel vorhanden ist, an den Anfang einer Liste ein Element einzufügen, nehmen wir die Liste des zweiten Argumentes von Anhaengen als Ausgangsbasis und betrachten das Problem als Einfügen der Liste des ersten Argumentes. Wir unterstellen wieder die Existenz einer Teillösung, nämlich das Verknüpfen der Restliste des ersten Argumentes mit der Liste des zweiten Argumentes. Die Lösung für das Gesamtproblem ist dann das Einfügen des ersten Elementes der ersten Liste in diese Teillösung. Das Einfügen übernimmt das cons-Konstrukt. Damit ist Anhaengen, wie folgt, definierbar:

```
eval> (define Anhaengen
        (lambda (Liste_1 Liste_2)
          (cond ((null? Liste_1) Liste_2)
                (#t (cons (car Liste_1)
                          (Anhaengen (cdr Liste_1)
                                     Liste_2))))))
eval> (Anhaengen '(1 2 3) '(4 5 6)) ==>
      (1 2 3 4 5 6)
```

Bisher wurde die Rekursion sowohl als eine iterativ geprägte Formulierung (*tail recursion*) als auch im Zusammenhang mit Verknüpfungsfunktionen wie cons oder append erörtert. Jetzt diskutieren wir die Rekursion im Hinblick auf geschachtelte Aufrufe (Komposition). Der rekursive Bezug wird als Argument eines schon rekursiven Funktionsaufrufes formuliert. Dazu dient wieder die Funktion Umdrehen; jedoch wird diese nicht mehr mittels append definiert.

Der Lösungsansatz geht von folgenden Überlegungen aus:

1. Enthält die umzudrehende Liste kein Element, dann ist die leere Liste () die Lösung.

2. Enthält die umzudrehende Liste nur ein Element, dann ist sie selbst die Lösung.

3. Ist die Liste x, z. B. (A B C D), umzudrehen, dann ist das letzte Element der Liste, hier D, an den Anfang einer Liste zu setzen, deren weitere Elemente die umgedrehte Restliste, hier das Umdrehen von (A B C), ist.

4. Das letzte Element einer Liste x ist selektierbar durch:

   ```
   eval> (car (Umdrehen (cdr x))
         ==> D ;letzte Element von x
   ```

5. Gesucht ist die rekursive Definition, welche die Liste x ohne ihr letztes Element, hier D, als Wert hat.

Die folgende rekursive Definition in LISP-Notation verdeutlicht den skizzierten Lösungsansatz:

$$(Umdrehen\ x) \equiv \begin{cases} () & falls \quad (null?\ x) \\ x & falls \quad (null?\ (cdr\ x)) \\ (cons < letzte-element > & falls \quad Sonst \\ \quad (Umdrehen & \\ \quad < liste-ohne-letzte- & \\ \quad\ element >)) & \end{cases}$$

Die Formulierung für den Sonst-Fall erarbeiten wir durch folgende Schritte, wobei für die Erläuterung angenommen wird, dass an `x` die Liste `(A B C D)` gebunden ist:

1. Restliste umdrehen:

```
eval> (Umdrehen (cdr x)) ==> (D C B)
```

2. Von Schritt 1 das erste Element entfernen:

```
eval> (cdr (Umdrehen (cdr x))) ==> (C B)
```

3. Ergebnis von Schritt 2 umdrehen

```
eval> (Umdrehen (cdr (Umdrehen (cdr x))))
      ==> (B C)
```

4. Erste Element von `x`, also `A`, in Ergebnis von Schritt 3 einfügen:

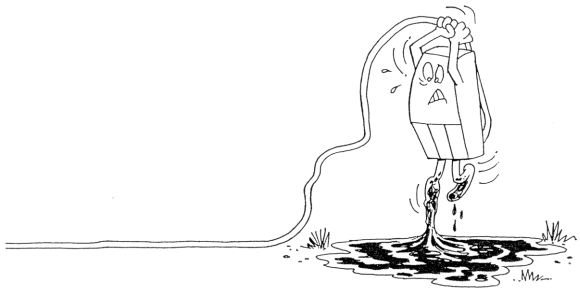
```
eval> (cons (car x)
            (Umdrehen (cdr
                        (Umdrehen (cdr x)))))
      ==> (A B C)
```

Die Definition von Schritt 4 ermittelt aus der Liste `(A B C D)` die Liste `(A B C)`. Damit ist entsprechend der skizzierten Lösung das Konstrukt `Umdrehen` wie folgt notierbar:

```
eval> (define Umdrehen
        (lambda (x)
          (cond ((null? x) null)
                ((null? (cdr x)) x)
                (#t (cons
                 (car (Umdrehen (cdr x)))
                 (Umdrehen (cons (car x)
```

```
                                    (Umdrehen
                                     (cdr
                                       (Umdrehen
                                         (cdr x))
                                     ))))))
                  )))
eval> (Umdrehen '(A B C D)) ==> (D C B A)
```

Diese Definition des Konstruktes `Umdrehen` zeigt geschachtelte rekursive Bezüge. Nur durch Rekursion und Abstützung auf die Basiskonstrukte `lambda`, `cond`, `cons`, `car`, `cdr`, `null` (leere Liste) und `null?` (Prüfung auf leere Liste) ist das Umdrehen abgebildet. Mit Hilfe der Rekursion lassen sich viele eingebaute Konstrukte auf wenige Basiskonstrukte, d. h. auf das sogenannte *Pure LISP*, zurückführen .



1.3.5 Zusammenfassung: Rekursion

Nimmt eine Definition auf sich selbst Bezug, so bezeichnet man diesen Selbstbezug als Rekursion. Die Lösung eines Problems durch Rekursion ist angebracht, wenn das Problem als eine Familie „verwandter“ Probleme betrachtbar ist, von denen eines (oder mehrere) so einfach ist (sind), dass sich die Lösung(en) direkt angeben lässt (lassen). Neben diesen trivialen Lösungen ist eine Rekursions-Route zu definieren, so dass die Lösung für das gesamte Problem nach endlich vielen Schritten auf die trivialen Lösungen zurückgeführt wird. Dabei geht es um die Kernfrage: Wie kommt man von einem typischen Fall genau einen Schritt näher zu einem trivialen Fall und wie konstruiert man für den typischen Fall die Lösung aus der „unterstellten“ Lösung für den nächst einfacheren Fall?

Besteht das Problem in der Abarbeitung einer Liste, z. B. im Prüfen, ob jedes Element ein bestimmtes Prädikat erfüllt, dann ist die Lösung als eine rekursive Anwendung der Funktion auf die Restliste definierbar. Ein Beispiel ist das `member`-Konstrukt, das feststellt, ob ein vorgegebenes Element in einer Liste enthalten ist (↪ charakteristische Beispiele S. 132).

Ist eine Baumstruktur gegeben, d. h. eine Liste, die wieder selbst Listen als Elemente enthält, dann sind die rekursiv definierten Teilproblem-

Lösungen miteinander zu verknüpfen. Dafür bieten sich die Konstruktoren `cons` und `append` an. Ein Beispiel ist die folgende Funktion `Turn`. Sie kehrt die Reihenfolge der Elemente einer Liste um, wobei auch die Elemente in den Sublisten umgedreht werden.

```
eval> (define Turn
         (lambda (Liste)
           (cond ((null? Liste) null)
                 ((not(pair? (car Liste)))
                  (append (Turn (cdr Liste))
                                (cons (car Liste)
                                      null)))
                 (#t (append (Turn (cdr Liste))
                             (cons (Turn (car Liste))
                                   null))))))
eval> (Turn '(A B (C D) E)) ==> (E (D C) B A)
```

Die Definition lokaler rekursiver Funktionen ermöglicht das `letrec`-Konstrukt. Im Gegensatz zum `let`-Konstrukt, das erst die Werte für alle lokalen Variablen ermittelt und dann mit diesen in der Umgebung Assoziationen aufbaut, vermerkt das `letrec`-Konstrukt zunächst die lokale Variable in der Umgebung und ermittelt dann ihren Wert.

```
eval> (letrec
          ((Foo (lambda (l)
                  (cond ((null? l) null)
                        ((= (car l) 0)
                         (cons "Undefiniert"
                               (Foo (cdr l))))
                        (#t (cons (/ 1 (car l))
                                  (Foo
                                   (cdr l))))))))
        (Foo '(2 0 4 0 1)))
     ==>
     (1/2 "Undefiniert" 1/4 "Undefiniert" 1)
```

Charakteristische Beispiele für Abschnitt 1.3

Beispiele: `member?` **und** `union`

```
eval> (define member?
        (lambda (Element Liste)
          (cond ((null? Liste) #f)
                ((equal? Element (car Liste))
                 Liste)
                (#t (member? Element
                             (cdr Liste))))))
eval> (member? 'A '(B C D)) ==> #f
eval> (member? 'B '(A B C D))
```

```
      ==> (B C D)
eval> (member? '(A B) '(A B C D))
      ==> #f
eval> (member? '(A B) '(A (A B) (C)))
      ==> ((A B) (C))

eval>  ;Rekursive Definition
       (define union-sequenz
        (lambda (Liste_1 Liste_2)
          (cond ((null? Liste_1) Liste_2)
                ((member? (car Liste_1) Liste_2)
                 (union-sequenz (cdr Liste_1)
                                Liste_2))
                (#t (cons (car Liste_1)
                          (union-sequenz
                            (cdr Liste_1)
                              Liste_2))))))
eval> (union-sequenz '(A A B B C)
                     '(B D D E))
      ==> (A A C B D D E)

eval> ; Endrekursive Definition
eval> (define union
        (lambda (Liste_1 Liste_2)
          (cond ((null? Liste_1) Liste_2)
                ((member? (car Liste_1) Liste_2)
                 (union (cdr Liste_1)
                        Liste_2))
                (#t (union (cdr Liste_1)
                           (cons (car Liste_1)
                                 Liste_2))))))
eval> (union '(A A B B C)
         '(B D D E))
      ==> (C A B D D E)
eval> (union '(A A B C)
             '(A B C D E))
      ==> (A B C D E)
```

In der endrekursiv-definierten `union`-Lösung arbeitet das Prädikat `member?` stets das erzeugte Zwischenresultat ab und nicht nur den Wert des ursprünglichen Argumentes `Liste_2`. Die folgende Lösung `union-menge` verfügt über eine lokale rekursive Funktion mit einer `lambda`-Variablen `Akkumulator`.

```
eval> (define union-menge
        (lambda (Liste_1 Liste_2)
         (letrec
          ((Vereinigung
            (lambda (L_1 L_2 Akkumulator)
              (cond ((and (null? L_1)
```

```
                          (null? L_2))
                     Akkumulator)
                    ((null? L_1)
                     (Vereinigung L_2 L_1
                                  Akkumulator))
                    ((member? (car L_1)
                              Akkumulator)
                     (Vereinigung (cdr L_1)
                                  L_2
                                  Akkumulator))
                    (#t (Vereinigung
                         (cdr L_1)
                         L_2
                         (cons (car L_1)
                               Akkumulator)))))))
        (Vereinigung Liste_1
                     Liste_2
                     null))))
eval> (union-menge '(A B) '(C D E))
      ==> (E D C B A)
eval> (union-menge '(A A B B C) '(B D D E))
      ==> (E D C B A)
```

Beispiel: Exponential-Funktion für ganzahlige Exponenten größer Null

$$x^n = \begin{cases} 1 & falls \quad n = 0 \\ (x^{\frac{n}{2}})^2 & falls \quad n \equiv gradzahlig(even?) \\ x * x^{(n-1)} & falls \quad n \equiv ungradzahlig \end{cases}$$

```
eval> (define **
        (lambda (Basis Integer_Zahl)
          (letrec ( ;;Prüfen auf gerade Zahl
                   (even?
                    (lambda (n)
                      (= 0 (remainder n 2))))
                   ;;Quadratfunktion
                   (square
                    (lambda (y) (* y y)))
                   ;;Exponential Funktion
                   (expt
                    (lambda (x n)
                      (cond ((= 0 n) 1)
                            ((even? n)
                             (square
                               (expt x (/ n 2))))
                            (#t (* x
                                   (expt x
```

```
                                        (- n 1))))))))
            (expt Basis Integer_Zahl))))
eval> (** 3 2) ==> 9
```

Beispiel: Vergleich zweier Listen Vergleich zweier Listen, wobei das Element mit dem numerischen Maximalwert der ersten Liste ermittelt wird. Rückgabewert ist das Element der zweiten Liste, das an der gleichen Position steht.

```
eval> ;;Iterative Lösung
      (define Maximum-Parallel-Wert
        (lambda (Zahlen_Liste Such_Liste)
          (do ((ZL Zahlen_Liste (cdr ZL))
               (SL Such_Liste (cdr SL))
               (ZL_Max 0)
               (SL_Max null))
            ((null? ZL) SL_Max)
            (cond ((< ZL_Max (car ZL))
                   (set! ZL_Max (car ZL))
                   (set! SL_Max (car SL)))
                  (#t #f)))))

eval> ;;Rekursive Lösung
      (define Maximum-Parallel-Wert
        (lambda (Zahlen_Liste Such_Liste)
          (letrec
              ((Max-Parallel
                (lambda (ZL SL ZL_Max SL_Max)
                  (cond ((null? ZL) SL_Max)
                        ((< ZL_Max (car ZL))
                         (Max-Parallel
                          (cdr ZL)
                          (cdr SL)
                          (car ZL)
                          (car SL)))
                        (#t (Max-Parallel
                             (cdr ZL)
                             (cdr SL)
                             ZL_Max
                             SL_Max))))))
            (Max-Parallel Zahlen_Liste
                          Such_Liste
                          0
                          null))))

eval> (Maximum-Parallel-Wert
        '(2 4 1 2 6)
        '(LISP ist manchmal sehr nuetzlich))
      ==> nuetzlich
```

```
eval> (Maximum-Parallel-Wert
        '(2 5 1 3)
        '(LISP ist manchmal sehr nuetzlich))
      ==> ist
```

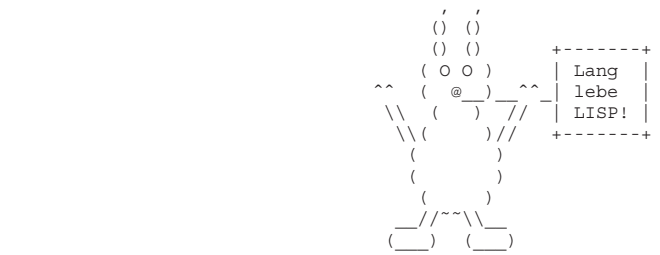
Beispiel: Ergänzung einer bestehenden Assoziationsliste

```
eval> (define Ergaenzung-A-Liste
        (lambda (Liste_Keys Liste_Werte A_Liste)
          (cond ((and (null? Liste_Keys)
                      (null? Liste_Werte))
                 A_Liste)
                ((null? Liste_Keys)
                 (Ergaenzung-A-Liste
                  (list '?)
                  Liste_Werte
                  A_Liste))
                ((null? Liste_Werte)
                 (Ergaenzung-A-Liste
                  Liste_Keys
                  (list '?)
                  A_Liste))
                (#t (cons
                     (list (car Liste_Keys)
                           (car Liste_Werte))
                     (Ergaenzung-A-Liste
                      (cdr Liste_Keys)
                      (cdr Liste_Werte)
                      A_Liste))))))
eval> (Ergaenzung-A-Liste
        '(Emma Hans Karl)
        '(verheiratet ledig)
        '((Otto verwitwet) (Hans ledig)))
      ==>
((Emma verheiratet) (Hans ledig) (Karl ?)
  (Otto verwitwet)  (Hans ledig))
```

Beispiel: Bestimmung der Schachtelungstiefe

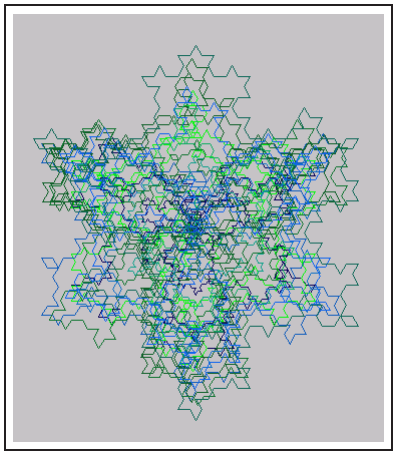
```
eval> (define Struktur-Tiefe
        (lambda (Liste)
          (cond ((null? Liste) 1)
                ((not (pair? Liste)) 0)
                (#t (+ (apply
                        max
                        (map
                         Struktur-Tiefe
                         Liste)) 1))
                )))
```

```
eval> (Struktur-Tiefe
        '((A B) C (((D (E F))))))
      ==> 5
```



Kapitel 2

Konstruktionen



Im Rahmen des zweiten Kapitels analysieren wir Programmierbeispiele und erweitern die Menge der im ersten Kapitel erläuterten Konstrukte („Bausteine“). Wir skizzieren charakteristische Konstruktionen („Bauarten“) und vertiefen diese im Zusammenhang mit (Programmier)-Paradigmen, d. h. mit Vorstellungen des Konstrukteurs über die Abarbeitung durch einen (möglicherweise auch fiktiven) Rechner. Im Mittelpunkt stehen die konstruktiven („handwerklichen“) Aspekte des Programmierens und weniger die ästhetischen, wie sie etwa im Leitmotto *The Art of Programming* zum Ausdruck kommen.

Hinweis: Begriff *Konstruktion*
Der Begriff Konstruktion (lateinisch: Zusammenschichtung) bezeichnet in der Technik die Bauart eines Produktes oder Erzeugnisses. In der Mathematik bezeichnet man damit die Herleitung oder Ermittlung eines mathematischen „Objektes“ mit Hilfe genau beschriebener Operationen (Konstruktionsvorschriften) aus bestimmten Ausgangsobjekten. Beispielsweise

> ist in der Geometrie eine gesuchte Figur aus den vorgegebenen Elementen wie Geraden, Winkeln und Kreisen herzuleiten. Die konstruktive (formale) Logik ist eine wissenschaftliche Disziplin (Theorie) des Argumentierens, Beweisens und Schließens. In der Linguistik ist das Konstruieren das Zusammenfügen von Satzgliedern, Wortgruppen und Wörtern nach Regeln der Grammatik. Die Konstruktion ist eine in Konstituenten zerlegbare, komplexe sprachliche Einheit.

Als Einstieg dient ein klassisches LISP-Beispiel: „Symbolisches Differenzieren“. Dabei zeigt sich der Vorteil, ein Programm als einen Verbund von Konstruktor, Selektor, Prädikat und Mutator zu konzipieren (↪ Abschnitt 2.1 S. 141).

Die Wahl der zweckmäßigen Abbildung von Daten, die sogenannte Datenrepräsentation, ist eine Kompromissentscheidung zwischen diversen, zum Teil konkurrierenden Anforderungen, z. B.: Effizienter Datenzugriff, Datenmodifizierbarkeit, Speicherplatzbedarf oder Lebensdauer. LISP verfügt daher, wie andere Programmiersprachen auch, nicht nur über eine Darstellungsmöglichkeit. Neben der Liste gibt es z. B. Vektoren, Zeichenketten, Ports und Ströme. Wegen der in LISP bestehenden Ambivalenz zwischen „Daten“ und „Programm“ (↪ Abschnitt 1.1.3 S. 27) sprechen wir hier allgemein von einer Abbildungsoption.

Zunächst vertiefen wir die Abbildungsoption Liste als Grundlage einer Konstruktion (↪ Abschnitt 2.2 S. 160). Durch Modifikation der `cons`-Zelle konstruieren wir eine zirkuläre Liste (↪ Abschnitt 2.2.1 S. 166). Darauf aufbauend werden die Konstrukte für die Assoziationsliste (↪ Abschnitt 2.2.2 S. 183) und die Eigenschaftsliste (↪ Abschnitt 2.2.3 S. 191) erörtert.

Anhand des relativ größeren Beispiels „Aktenverwaltung“ diskutieren wir Konstrukte für die Abbildungsoption Vektor (↪ Abschnitt 2.3 S. 215). Die Option Zeichenkette erörtern wir im Rahmen der Aufgabe eines Mustervergleichs (↪ Abschnitte 2.4.1 S. 239 und 2.4.2 S. 241). Wir befassen uns mit dem `print`-Namen eines Symbols als eine weitere Abbildungsoption (↪ Abschnitt 2.4.3 S. 248). Erklärt wird die Notwendigkeit ein neues Symbol als Unikat zu generieren (↪ Abschnitt 2.4.4 S. 259). Dabei wird die Nützlichkeit des Mustervergleichs am Beispiel eines kleinen Vorübersetzers für eine Pseudocode-Notation demonstiert.

Ausführlich wird die Funktion und ihre Umgebung als Abbildungsoption behandelt (↪ Abschnitt 2.5 S. 270). Das Konstruieren von Funktionen höherer Ordnung erörtern wir anhand der beiden Beispiele: Abbildung der `cons`-Zelle als Funktion (↪ Abschnitt 2.5.1 S. 272) und Rekursion mit dem Fixpunktoperator Y (↪ Abschnitt 2.5.1 S. 277).

Wir können in Scheme nicht nur die Funktion selbst an ihre eigene `lambda`-Variable binden, sondern mit dem `call/cc`-Konstrukt auch die noch nicht vollzogene Abarbeitung. Mit dieser sogenannten Continuation (Fortsetzung) konstruieren wir Kontrollstrukturen, die als Sprünge in die „Zukunft“ und in die „Vergangenheit“ einer Berechnung erklärbar

sind (↪ Abschnitt 2.5.2 S. 285).

Die Möglichkeiten der Syntaxverbesserung durch Makros zeigen wir am Beispiel `definep`, einem Konstrukt, das mehrere Symbole binden kann (↪ Abschnitt 2.5.3 S. 293). Die in Kapitel I skizzierte Kontenverwaltung verwenden wir, um Schritt für Schritt die „Objektprägung“ von Konstruktionen zu steigern. Zur Abschirmung von Wertebeschreibungen (↪ Abschnitt 2.6.1 S. 306) dient zunächst das `define-struct`-Konstrukt (↪ Abschnitt 2.6.2 S. 309). Aufbauend auf die damit realisierbare einfache statische Vererbung (↪ Abschnitt 2.6.3 S. 314) wird die Abbildungsoption „Klasse-Instanz“ erklärt (↪ Abschnitt 2.8 S. 327). Eine „Klasse-Instanz“-Kontenverwaltung zeigen wir mit dem *Object-Oriented Programming System* von *PLT-Scheme* [1] (↪ Abschnitt 2.8.1 S. 329). Es erlaubt vielfältige Optionen der Vererbung (↪ Abschnitt 2.8.2 S. 337).

2.1 Verstehen einer Konstruktion

Wollen wir Konstruktionen (Programme) verstehen, dann sind die einzelnen Konstrukte und ihre Verknüpfungen (Wechselwirkungen) untereinander zu analysieren. Da die verfolgten Analysezwecke vielfältig sind, müssen Konstrukte im Hinblick auf unterschiedliche Aspekte hinreichend verstanden werden. Die Konstrukte können z. B. nach ihrer Aufgabe, ihrem Leistungsumfang oder ihrer Abarbeitung durch das LISP-System untersucht werden (↪ Abschnitt 2.1.1 S. 145).

Eine derartige Klassifikation zeigt Tabelle 2.1 S. 142. So gehört z. B. das `do`-Konstrukt (↪ S. 90) in Scheme zur Klasse der „eingebauten Konstrukte“. Als Benutzer können wir es der Klasse „spezielle Konstrukte“ zuordnen, weil es seine Argumente nach einer speziellen Vorschrift evaluiert. Aus der Aufgabensicht fällt es in die Klasse „Steuerungskonstrukte“, weil es einen Wiederholungsprozess (Iteration) steuert. Aus der Implementationssicht ist es oft ein Makro, das zu einem `letrec`-Konstrukt expandiert.

Jede Konstruktion ist geprägt von den Vorstellungen, die der (Chef-)-Konstrukteur hat, wenn er über die Ausführung durch einen Rechner nachdenkt. Für sein Ausführungsmodell sind spezifische Konstrukte, Verknüpfungen, Lösungsansätze und Vorgehensweisen charakteristisch. Ein Modell orientiert sich an beispielhaften, besonders gelungenen Konstruktionen. Sie bilden den Rahmen in dem der Konstrukteur seine Lösungen sucht, notiert und bewertet.

Ein derartiger „Denkrahmen“ wird als Programmierparadigma oder kurz als Paradigma[2] (≈ Denkmodell), bezeichnet. Man könnte weni-

[1] *PLT-Scheme* ↪`http://www.plt-scheme.org/` (Zugriff: 24-Oct-2009)

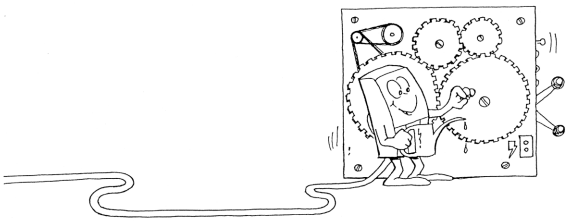
[2] Ein Paradigma ist ein von der wissenschaftlichen Fachwelt als Grundlage der weiteren Arbeiten anerkanntes Erklärungsmodell, eine forschungsleitende Theorie. Es entsteht, weil es bei der Lösung von als

Klassifikation für Konstrukte

Analyseaspekt	Konstruktmerkmal	Beispiele bzw. Erläuterung
Ersteller des Konstruktes	eingebaut	`cons`, `car`, `cdr`, `null`, `eq?`, `pair?`
	selbst definiert	`Konto`, `Abbuchen!`
Abarbeitung durch das LISP-System	normal	`equal?`, `+`, `-`
	speziell	`if`, `quote`, `lambda`, `define`
Leistungsumfang	primitiv	`null?`
	komplex	`do`, `case`, `letrec`
Verwendungsfehlerrisiko	problemlos	`cons`, `list`
	destruktiv	`append!`,`set-car!`, `set-cdr!`
Reichweite	global	Definiert in USER-GLOBAL-ENVIRONMENT bzw. USER-INITIAL-ENVIRONMENT
	lokal	`do`, `lambda`, `let`
Aufgabe	Konstruktor	`cons`, `Make-Konto`
	Selektor	`car`, `Summe-Einzahlungen`
	Prädikat	`null?`, `zero?`, `Konto`
	Mutator	`set!`, `Einzahlen!`
Ablauf	Steuerung (*transfer of control*)	Sequenz, Selektion, Iteration, `goto`,`exit`
	Umgebungsoperation im System	Bindung von lokalen Variablen
	Kommunikation mit der Umwelt	`read`, `print`, `load`

Tabelle 2.1: Klassifikation für Konstrukte

ger anspruchsvoll auch von einem (Programmier-)Stil reden, wenn wir dabei die jeweils bestimmenden Stilelemente in den Mittelpunkt rücken (↪ [176]). Allgemein ist ein Paradigma ein von der wissenschaftlichen Fachwelt als Grundlage der weiteren Arbeiten anerkanntes Erklärungsmodell, eine forschungsleitende Theorie.



Unerheblich ist, ob das Modell direkt von einem „real existierenden" Rechner abgearbeitet werden kann, oder ob es auf einem fiktiven Rechner (einer angenommenen Maschine) basiert. Im letzteren Fall lässt sich der Konstrukteur nicht davon leiten, wie die Umsetzung seines Modells auf einen verfügbaren Rechner vollzogen wird. Er unterstellt nur, dass seine paradigma-spezifische Kombination von Eigenschaften, Merkmalen und Besonderheiten für seine Konstruktions-Aufgabe zweckmäßig und prinzipiell auch effizient ist. Praxisrelevant ist für ihn die Frage: Existiert eine paradigma-konforme Entwicklungsumgebung, welche die spezifischen Konstrukte direkt bereitstellt und ihre Verknüpfung unterstützt?

Angenommen, es sei eine Konstruktion zu analysieren, bei der sich der Konstrukteur einen Rechner vorgestellt hat, der Objekte generieren und modifizieren kann, die miteinander Nachrichten austauschen. In diesem objekt-geprägten Modell werden Abstraktionen durch Klassen- und Instanz-Definitionen kombiniert mit Vererbungsmechanismen abgebildet. Es dominieren daher Konstrukte zur Definition und Modifikation von anwendungsspezifischen Klassen und Instanzen (Näheres ↪ Abschnitt 2.8 S. 327). Wäre das gleiche Problem mit einem funktionsgeprägten Modell gelöst, dann dominierten anwendungsspezifische Funktionen, wobei Abstraktionen durch Funktionen höherer Ordnung abgebildet wären (Näheres ↪ Abschnitt 2.5 S. 270). Offensichtlich erleichtern Kenntnisse über den angewendeten Denkrahmen das Verstehen einer vorgegebenen Konstruktion.

Der Denkrahmen, in welchem wir ein Problem analysieren und unsere Ideen für seine Lösung gewinnen und notieren, umfasst zumindest:

1. *elementare Konstrukte*
   Diese Basiskonstrukte (engl.: *primitives*) stellen die elementaren Einheiten dar, mit denen der Konstrukteur umgehen kann.

dringlich erkannten Problemen erfolgreicher ist (zu sein scheint) als andere, bisher „geltende" Ansätze (Kritik am klassischen Paradigma der Softwareentwicklung ↪ [22]).

2. *Verknüpfungsmittel*
   Sie ermöglichen das Konstruieren von komplexen Konstrukten aus weniger komplexen.

3. *Abstraktionsmittel*
   Sie benennen komplexe Konstrukte, um diese dann als Einheiten handhaben (manipulieren) zu können.

Moderne LISP-Systeme unterstützen verschiedene Denkrahmen (Paradigmen). Sie stellen zumindest Mittel bereit für:

1. imperativ-geprägte Programme (z. B. `set!`)
   Kennzeichnend sind die Erklärungen von Variablen und die schrittweise Änderung ihrer Werte durch Zuweisungsbefehle (engl.: *assign statements*). Diese Prägung unterstützen z. B. die Programmiersprachen *COBOL*[3] oder *Pascal*.

2. funktions-geprägte Programme (z. B. `lambda`, `apply`)
   Kennzeichnend sind Kompositionen von Funktionen höherer Ordnung. Diese Prägung unterstützt z. B. die Programmiersprache *ML* (↪ [75]).

3. objekt-geprägte Programme (z. B. `class`, `make-instance`)
   Kennzeichnend sind Objekte (lokale Speicher), deren Eigenschaften durch einen Vererbungsmechanismus bestimmt werden und die Nachrichten untereinander austauschen. Diese Prägung unterstützen z. B. die Programmiersprachen *SmallTalk*, *C++* und Flavor (↪ Abschnitt 2.8 S. 327).

Hinweis: *Orientiert*, *basiert* oder *geprägt*.
In der Literatur werden in diesem Zusammenhang Beiworte wie „orientiert“ oder „basiert“ verwendet. Stets verdeutlichen solche Beiworte, dass der Denkrahmen nicht auf die reine Lehre beschränkt ist. Wir bezeichnen eine Konstruktion als funktions-geprägt, selbst wenn sie z. B. das `print`-Konstrukt enthält, das wegen des Nebeneffektes eingesetzt wird und nicht zur Erzielung eines Funktionswertes. Entscheidend ist das „Sich-leitenlassen“ vom funktionalen Modell. Kompositionen anwendungsspezifischer Funktionen, die (weitgehend) frei von Nebeneffekten definiert sind, geben der Konstruktion ihr Beiwort *funktions-geprägt*.

Die Folge dieser LISP-„Potenz“ ist eine Konstruktevielfalt. Um sich in dieser großen Menge zurechtzufinden, bedarf es eines pragmatischen Leitfadens (↪ Abschnitt 2.1.2 S. 146). Wir wählen dazu eine aufgabenbezogene Klassifikation in die vier Typen:

1. Konstruktor,

2. Selektor (incl. Präsentator),

[3] ↪ S. 11.

3. Prädikat und

4. Mutator, auch Modifikator genannt (incl. Destruktor).

Ein Konstruktor erzeugt und kombiniert „Einheiten“ zu neuen „Einheiten“. Dabei ist eine „Einheit“ z. B. ein Objekt, eine Funktion, Menge, Liste oder ein Vektor. Ein Selektor ermöglicht den Zugriff auf eine solche „Einheit“. Prädikate vergleichen „Einheiten“. Sie prüfen z. B. auf Gleichheit oder Identität und stellen damit den „Typ“ fest. Mutatoren ändern existierende „Einheiten“. In dieses pragmatische, enge Klassifikationsschema integrieren wir auch Vorgänge, wie das Zerstören von „Einheiten“ (Destruktor hier als Mutator klassifiziert) oder das Präsentieren von „Einheiten“ (Präsentator hier als Selektor klassifiziert).

Der zeitliche Aspekt z. B. das Aktivieren, Terminieren und Suspendieren von „Einheiten“ ist damit kaum erfassbar. Wir decken diesen Aspekt daher durch Graphiken (z. B. Vererbungsgrafen) und zusätzliche Erläuterungen ab.

Gezeigt wurde, dass eine strikte begriffliche Trennung in Programm (≡ Algorithmus) und Daten(struktur) die Ambivalenz zwischen beiden verdeckt (↪ Abschnitt 1.1.3 S. 27). Zum Verstehen von LISP-Programmen erscheint es sinnvoll, diese Trennlinie nicht permanent zu betonen. Wir sprechen daher im Folgenden weniger von Programm- und Datenstrukturen sondern von Optionen zur Abbildung von „Einheiten“.

2.1.1 Analyseaspekte

Der Zweck einer Analyse bestimmt die Suche nach den bedeutsamen (relevanten) Konstrukten und ihrem Zerlegungsgrad. Dabei ist ein Konstrukt entweder ein „bekanntes“ Konstrukt, oder es setzt sich wiederum aus zu analysierenden Konstrukten zusammen. Wir unterstellen, dass der Leser des Quellcodetextes zumindest die eingebauten LISP-Konstrukte erkennt und ihm ihre Wirkung (gegebenenfalls durch Nachschlagen im jeweiligen Referenzmanual) hinreichend bekannt ist.

Von den mannigfaltigen Analysezwecken sind z. B. zu nennen:

1. Das Nachvollziehen von Ergebnissen (Output) bei vorgegebener Ausgangssituation (Input),

2. das Prüfen der Übereinstimmung mit der Spezifikation,

3. die Erweiterung des Leistungsumfangs,

4. die Korrektur von Fehlern,

5. die Veränderung des Ressourcen-Bedarfs (Stichwort: *Tunen*) und

6. die Portierung auf ein anderes System.

Das Herausfinden und Verstehen der Konstrukte, die für den jeweiligen Analysezweck relevant sind, ist keine triviale Aufgabe. Zu beantworten sind die beiden Fragen:

1. Aus welchen bekannten Konstrukten besteht die Konstruktion?
2. Welche Wechselwirkungen (Interaktionen) bestehen zwischen diesen Konstrukten?

Als Beispiele analysieren wir zwei Programmfragmente (↪ Tabelle 2.2 S. 147 und 2.3 S. 148) im Hinblick auf eine Portierung nach Scheme. Beim ersten Programmfragment ist die Beseitigung der Namensunterschiede von eingebauten Konstrukten (hier: `ZEROP` bzw. `zero?`) nicht ausreichend. Zu berücksichtigen sind die unterschiedlichen Bindungsstrategien des namensgleichen Konstruktes `let`. Etwas salopp formuliert: *Kein Verlass auf Namen!* Im zweiten Programmfragment ist die Iteration auf direkte Weise durch einen expliziten Rücksprung mit dem `GO`-Konstrukt und der Sprungmarke `LOOP` realisiert. Das Verlassen der Iteration geschieht über das `RETURN`-Konstrukt. Diese Aufgabe kann in Scheme entweder ein `do`-Konstrukt oder eine rekursive lokale Funktion übernehmen.

Für das Verstehen zum Zweck der Portierung mag das Durcharbeiten der Quellcodetexte, beginnend von der ersten Zeile bis zum Schluss, erfolgreich sein. Handelt es sich jedoch um ein komplexes (Programm-)-System, dann ist für das Verstehen die sequentielle Betrachtung „Von-Anfang-bis-Ende-des-Textes“ selten hilfreich. Zumindest sind in der alltäglichen Programmierpraxis schon aus Zeitgründen nicht alle Konstrukte in Notationsreihenfolge analysierbar. Das Verfolgen aller möglichen „Pfade“, die sich aufgrund der Steuerungskonstrukte ergeben, scheidet wegen ihrer großen Zahl aus. Die Bewältigung der Komplexität bedingt, dass wir uns an Abstraktionsebenen orientieren.

2.1.2 Verbund von Konstruktor, Selektor, Prädikat und Mutator

Ein komplexes Konstrukt setzt sich aus einfacheren Konstrukten zusammen. In der Abbildung 2.1 S. 149 besteht das Konstrukt $K-1^0$ aus den Konstrukten $K-2^1$ und $K-3^1$, wobei der Exponent hier zur Kennzeichnung des Zerlegungsgrades dient.

Die analytische Betrachtung ist durch eine an der Synthese orientierten Betrachtung zu ergänzen. Das Konstrukt $K-1^0$ ist dann ein Konstruktor, der die Konstrukte $K-2^1$ und $K-3^1$ zusammenfügt. Damit ein Konstruktor die richtigen „Einheiten“ zusammenfügen kann, müssen die einzelnen „Einheiten“ erkannt werden. Benötigt werden daher passende Erkennungskonstrukte (engl.: *recognizer*). Erforderlich ist in unserem Beispiel ein Konstrukt, das feststellt, ob es sich bei einer gegebenen

```
;;;;TLC-LISP Lösung zum Feststellen,
;;;; ob eine Zahl N gerade ist.
T-eval> (LET ((GERADE?
                (LAMBDA (N)
                  (COND((ZEROP N) T)
                        (T (UNGERADE? (SUB1 N))))))
              (UNGERADE?
                (LAMBDA (N)
                  (COND((ZEROP N) NIL)
                        (T (GERADE? (SUB1 N)))))))
              (GERADE? 44)) ==> T

eval> ;;;;Lösung portiert nach Scheme
     (define zerop zero?) ;Namensangleichung
eval> ;;Portierungsproblem:
      ;; - TLC-LISP: dynamische Bindung
      ;; - SCHEME: statische Bindung
      (letrec  ;Hier letrec nicht let !!!!!!
          ((gerade?
            (lambda(n)
              (cond((zerop n) #t)
                   (#t (ungerade? (sub1 n))))))
           (ungerade?
            (lambda (n)
              (cond ((zerop n) #f)
                    (#t (gerade? (sub1 n)))))))
        (gerade? 44)) ==> #t
```

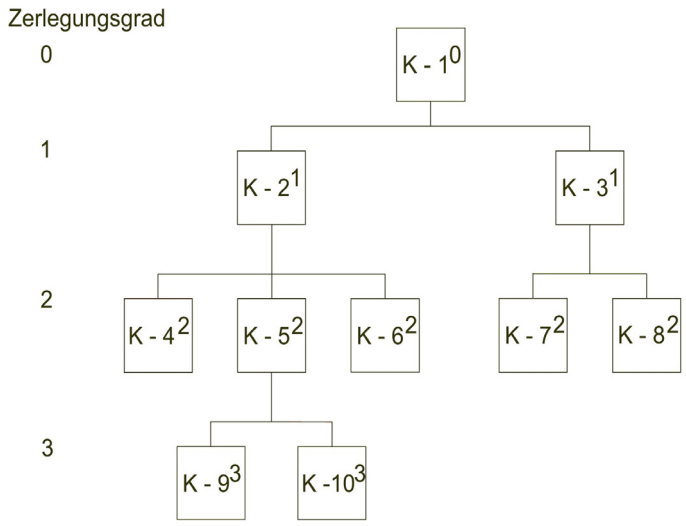
Legende:
TLC-LISP ↪ [183]

Tabelle 2.2: Programmfragment: Portierung von *TLC-LISP* nach *Scheme*

```
;;;;Iteration mittels "go to"- Konstrukt
T-eval> (PROG (X Y)       ;PROG definiert X und Y
              (SETQ X 10) ; als lokale Variablen
              (SETQ y 0)
         LOOP (PRINT (TIMES Y Y))
              (SETQ X (SUB1 X))
              (SETQ Y (ADD1 Y))
              (cond((ZEROP X)
                    (RETURN Y)) ;Verlassen der
                                ; Iteration
                    (T (GO LOOP))))   ==>
0
1
4 9 16 25 36 49 64 ; jede Zahl auf neuer Zeile
81 10
;;;;Lösung portiert nach Scheme; Portierungsproblem:
;;;  Das Erkennen und Verstehen der
;;;  "go to"-Konstruktion  und die Auswahl
;;;  der zweckmäßigen Lösung in Scheme.
eval> ;;Lösung mit do-Konstrukt
      (do ((x 10 (sub1 x))
           (y 0 (add1 y)))
        ((= x 0) y)
        (print (* y y))
        (newline)) ==> 0 ... 81 10
eval> ;;Alternative end-rekursive Funktion
     (letrec ((Foo (lambda (x y)
                     (cond ((= x 0) y)
                           (#t (print (* y y))
                               (newline)
                               (Foo (sub1 x)
                                    (add1 y)))))))
    (Foo 10 0)) ==> 0 ... 81 10
```

Legende:
Idee ↪ [180] S. 32.

Tabelle 2.3: Programmfragment: Portierung von einem kleinen LISP-System nach *Scheme*



Legende:

K-i^j	≡ Konstrukt mit der Identifizierungsnummer *i* auf der Abstrakionsebene j bzw. dem Zerlegungsgrad j.
—	≡ „Setzt-sich-zusammen“, z. B. setzt sich K-5^2 aus K-9^3 und K-10^3 zusammen.

Eine Konstrukte-Hierarchie wird auch als eine Modul-Hierarchie bezeichnet.

Abbildung 2.1: Skizze einer Konstrukte-Hierarchie

„Einheit“ um $K-2$ handelt oder nicht. Ein solches Erkennungskonstrukt verhält sich wie ein Prädikat wie z. B. das `number?`-Konstrukt, das prüft, ob ein numerischer Wert vorliegt.

Zu einem Konstruktor, der n-verschiedene „Einheiten“ zusammenfügt, benötigen wir n-verschiedene Prädikate zum Erkennen der einzelnen „Einheit“. Wir definieren daher für unseren Konstruktor `K-1` die korrespondierenden Prädikate `K-2?` und `K-3?`. Auf der nächst tieferen Abstraktionsebene, hier Ebene 1, wären z. B. für den Konstruktor `K-3` die Prädikate `K-7?` und `K-8?` zu definieren.

Zusätzlich benötigen wir korrespondierende Selektoren, um bei einem zusammengesetzten Konstrukt auf seine einzelnen „Einheiten“ zugreifen zu können. Die Selektoren haben die Aufgabe der Umkehrung der Konstruktor-Wirkung. Für den Konstruktor `K-1` in Abbildung 2.1 S. 149 wären folgende Selektoren zu definieren:

```
eval> (Selektor-K-2 K-1) ==> K-2
eval> (Selektor-K-3 K-1) ==> K-3
```

Die Analyse und damit das Verstehen werden erleichtert, wenn wir

von folgenden Eigenschaften bei unserem Verbund aus Konstruktoren, Selektoren und Prädikaten ausgehen können:

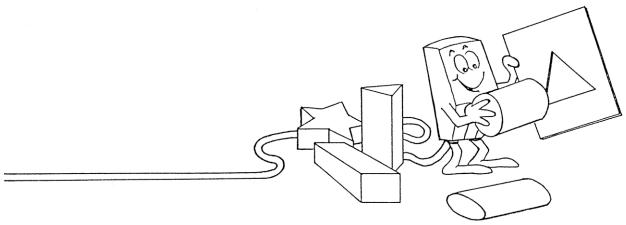
```
eval> (Selektor-K-2 (Konstruktor-K-1 K-2 K-3))
      ==> K-2
eval> (Selektor-K-3 (Konstruktor-K-1 K-2 K-3))
      ==> K-3
eval> (K-2? (Selektor-K-2
              (Konstruktor-K-1 K-2 K-3))) ==> #t
eval> (K-3? (Selektor-K-3
              (Konstruktor-K-1 K-2 K-3))) ==> #t
```

Wir streben daher für alle Abstraktionsebenen diese Verbundeigenschaften an. Sie sind allgemein, wie folgt, beschreibbar:

```
eval> (< selektor − bauteil_i >
        (< konstruktor >< bauteil_1 > ... < bauteil_n >))
      ==>< bauteil_i >

eval> (< praedikat − bauteil_i? >
        (< selektor − bauteil_i >
          (< konstruktor >< bauteil_1 > ... < bauteil_n >))
      ==> #t
```

Der Verbund von Konstruktoren, Selektoren und Prädikaten ermöglicht Neukonstruktionen. Soll eine bestehende Konstruktion geändert, also nicht von Anfang an neu zusammengesetzt werden, dann sind Modifikationskonstrukte, sogenannte *Mutatoren*, erforderlich. Betrachten wir z. B. ein Konto, von dem ein Geldbetrag abzubuchen ist, dann wollen wir (wahrscheinlich) mit einem Konstrukt `Abbuchen!` das bestehende Konto im Saldo ändern, d. h. modifizieren und nicht ein ganz neues Konto anlegen und alle alten Buchungen rekonstruieren und dann die neue Buchung zusätzlich vollziehen (↪ Beispiel S. 42).

In den folgenden Ausführungen dient dieser Verbund als pragmatischer Leitfaden. Er ist kein Patentrezept („Allheilmittel“). Er ersetzt keinesfalls intensives Durchdenken einer Konstruktion. Er hilft jedoch eine komplexe Aufgabe zweckmäßig in weniger komplexe Teile zu gliedern.



2.1.3 LISP-Klassiker: Symbolisches Differenzieren

Als Analysebeispiel betrachten wir das Differenzieren einer mathematischen Funktion anhand des Programms `Ableitungsfunktion` (↪ S. 153). Diese Aufgabenstellung ist ein LISP-Klassiker. Sie war das Markenzeichen beim Start der LISP-Entwicklung. Sie ist für Erläuterungszwecke weiterhin zweckmäßig (↪ z. B. [1] p. 104; [135] S. 13ff; [187] S. B-1ff, [189] p. 16 , oder [190], deutsche Ausgabe S. 276 ff). Wir nutzen den Vorzug, dass sie einerseits in LISP einfach (rekursiv) lösbar ist und andererseits keiner langen Erläuterung bedarf (Schulmathematik, ↪ z. B. [161]).

Das Programm `Ableitungsfunktion` (↪ Strukturskizze S. 157 und Programmcode S. 153) fasst die lokal definierten Konstruktoren, Selektoren und Prädikate zusammen. Es hat folgende Struktur:

```
eval> (define Ableitungsfunktion
  (lambda (Mathematischer_Ausdruck dx)
    (letrec
    ;;;;Prädikate für Konstante und Variable
     ((Konstante? ... )
      (Variable?  ... )
      (Gleiche-Variable? ... )
    ;;;;Prädikate, Selektoren und Konstruktoren
    ;;;; für die Summenregel
      (Summe? ... )
      (Erster-Summand ... )
      (Zweiter-Summand ...)
      (Bilde-Summe ...)
      (Vereinfache-Summe ... )
    ;;;;Prädikate, Selektoren und Konstruktoren
    ;;;; für die Produktregel
      (Produkt? ... )
      (Multiplikand ... )
      (Multiplikator ... )
      (Bilde-Produkt ... )
      (Vereinfache-Produkt ... )
    ;;;;Ableitungsregeln
      (Ableiten (lambda (Formel Variable)
          (cond...)))) ;Fallunterscheidung für die
                       ; anzuwendende Ableitungsregel
    ;;;;Anwendung der lokalen Funktion
   (Ableiten Mathematischer_Ausdruck dx))))
```

Beim Aufruf des Konstruktes `Ableiten` ist die `lambda`-Variable `Formel` an den abzuleitenden mathematischen Ausdruck gebunden. Mit Hilfe der Prädikate `Summe?` und `Produkt?` wird erkannt, ob es sich um eine Summe oder ein Produkt handelt. In diesen Fällen ist der Wert von `Formel` ein zusammengesetztes Konstrukt, dessen einzelne Teile zur Anwendung der entsprechenden Ableitungsregel benötigt werden. Im Fall

der Summe sind deshalb die Selektoren `Erster-Summand` und `Zweiter-Summand` definiert. Für das Produkt sind es die Selektoren `Multiplikand` und `Multiplikator`. Die Benennung der lokalen Funktionen und Kommentierung entsprechen den Empfehlungen von Abschnitt 3.1 S. 373.

Zum Verstehen der Aufgabe sind vorab Anforderungen und Testfälle formuliert. Dabei verweist die Abkürzung `A` auf eine *Anforderung* und die Abkürzung `T` auf einen *Testfall*.

`A1`: Die `Ableitungsfunktion` ermittelt die Ableitung einer mathematischen Formel nach einer vorgegebenen Variablen.

`A2`: Für dieses symbolische Differenzieren gelten die folgenden Ableitungsregeln:

`A2.1`: Ableiten einer Konstanten: $\frac{dc}{dx} = 0$

`A2.2`: Ableiten einer Variablen:

$$\frac{du}{dx} = \begin{cases} 1 & falls \quad u = x \\ 0 & falls \quad sonst \end{cases}$$

`A2.3`: Summenregel: $\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx}$

`A2.4`: Produktregel: $\frac{d(u*v)}{dx} = (\frac{du}{dx}) * v + u * (\frac{dv}{dx})$

`A3`: Die abzuleitende Formel ist in Präfixnotation anzugeben.

`T1`: Beispiel: $\frac{d(ax^2+bx+c)}{dx} = 2ax + b$

```
eval> (ableitungsfunktion
        '(+ (+ (* a (* x x)) (* b x)) c)
        'x)
      ==> (+ (* a (* 2 x)) b)
```

Wert der Beispielfunktion an der Stelle $x = 1$ mit $a = 3$ und $b = 5$

```
eval> (ableitungswert
        '(+ (+ (* a (* x x)) (* b x)) c)
         'x 1)
==> 11
```

Programm: Symbolisches Differenzieren

```
(define Ableitungsfunktion
 (lambda (Mathematischer_Ausdruck dx)
  (letrec
     ((Konstante? (lambda (x) (number? x)))
      (Variable? (lambda (x) (symbol? x)))
      (Gleiche-Variable?
       (lambda (Var_1 Var_2)
         (and (Variable? Var_1)
              (Variable? Var_2)
              (eq? Var_1 Var_2))))
;;;;Konstruktor, Selektoren und Prädikat
;;;; für die Summe
      (Summe?
       (lambda (x)
         (and (pair? x)
              (or (eq? (car x) '+)
                  (eq? (car x) 'Plus)))))
      (Erster-Summand (lambda (Summe)
                        (list-ref Summe 1)))
      (Zweiter-Summand (lambda (Summe)
                         (list-ref Summe 2)))
;;;Konstruktoren für die "rechte"
;;; Seite der Summenregel
      (Bilde-Summe
       (lambda (S_1 S_2)
         (list '+ S_1 S_2)))
      (Vereinfache-Summe
       (lambda (Summe)
         (let ((S-1 (Erster-Summand Summe))
               (S-2 (Zweiter-Summand Summe)))
           (cond((and (number? S-1) (number? S-2))
                    (+ S-1 S-2))
                ((and (number? S-1)
                      (zero? S-1)) S-2)
                ((and (number? S-2)
                      (zero? S-2)) S-1)
                ((Gleiche-Variable? S-1 S-2)
                  (Bilde-Produkt 2 S-1))
                 (#t Summe)))))
;;;;Konstruktor, Selektoren und Prädikat
;;;; für das Produkt
      (Produkt?
       (lambda (x)
         (and (pair? x)
              (or (eq? (car x) '*)
                  (eq? (car x) 'Mul)))))
      (Multiplikand (lambda (Produkt)
                      (list-ref Produkt 1)))
      (Multiplikator (lambda (Produkt)
                       (list-ref Produkt 2)))
;;;Konstruktoren für die "rechte" Seite
;;; der Produktregel
      (Bilde-Produkt
```

```
       (lambda (M_1 M_2)
         (list '* M_1 M_2)))
      (Vereinfache-Produkt
       (lambda (Produkt)
         (let ((M-1 (Multiplikand Produkt))
               (M-2 (Multiplikator Produkt)))
           (cond((or (and (number? M-1)
                          (zero? M-1))
                     (and (number? M-2)
                          (zero? M-2)))  0)
                ((eq? M-1 1) M-2)
                ((eq? M-2 1) M-1)
                ((and (number? M-1) (number? M-2))
                                    (* M-1 M-2))
                (#t Produkt)))))
;;;;Ableitungsregeln
      (Ableiten (lambda (Formel Variable)
         (cond((Konstante? Formel) 0) ;dc/dx ==> 0
               ((Variable? Formel)
                (cond((Gleiche-Variable? ;dx/dx ==> 1
                                  Formel Variable) 1)
                      (#t 0)))
               ((Summe? Formel) ;Summenregel
                (Vereinfache-Summe
                   (Bilde-Summe
                     (Ableiten (Erster-Summand Formel)
                               Variable)
                     (Ableiten (Zweiter-Summand Formel)
                               Variable))))
                ((Produkt? Formel) ;Produktregel
                (Vereinfache-Summe
                   (Bilde-Summe
                      (Vereinfache-Produkt
                         (Bilde-Produkt
                            (Ableiten
                              (Multiplikand Formel)
                              Variable)
                            (Multiplikator Formel)))
                      (Vereinfache-Produkt
                         (Bilde-Produkt
                            (Multiplikand Formel)
                            (Ableiten
                              (Multiplikator Formel)
                              Variable))))))
               (#t (print "Keine Ableitung fuer: ")
                   (print Formel)
                   (print " definiert"))))))
    ;;Startfunktion
    (Ableiten Mathematischer_Ausdruck dx))))
```

Die Applikation von `Ableitungsfunktion` erzielt damit folgende Ausgabe:

```
eval> (Ableitungsfunktion
```

```
        '(+ (+ (* a (* x x)) (* b x)) c) 'x)
      ==> (+ (* a (* 2 x)) b)
```

Zur Berechnung eines Wertes der `Ableitungsfunktion` definieren wir die folgende Funktion `Ableitungswert`:

```
eval> (define Ableitungswert
        (lambda (Mathematischer_Ausdruck
                 dx
                 an_der_Stelle)
          ((eval
            (list 'lambda
                  (list dx)
                  (Ableitungsfunktion
                    Mathematischer_Ausdruck
                   dx)))
           an_der_Stelle)))
eval> (define a 3)
eval> (define b 5)
eval> (Ableitungswert
       '(+ (+ (* a (* x x)) (* b x)) c) 'x 1)
      ==> 11
```

Erläuterungen: Die Ableitungsregel für eine Summe (kurz: Summenregel) lautet: $\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx}$. Das Ergebnis, die rechte Seite der Summenregel, ist ein zusammengesetzter Ausdruck, der wieder eine Summe darstellt. Das Zusammenfügen von $\frac{du}{dx}$ mit $\frac{dv}{dx}$ übernimmt der Konstruktor `Bilde-Summe`. Entsprechendes gilt für die Produktregel: $d(u*v) = (\frac{du}{dx})*v+u*(\frac{dv}{dx})$ Hier setzt der Konstruktor `Bilde-Produkt` in Kombination mit `Bilde-Summe` das Ergebnis zusammen.

Nehmen wir als Beispiel an, dass `S_1` an den Wert 5 und `S_2` an den Wert 7 gebunden sind, so dass gilt:

```
eval> (Bilde-Summe S_1 S_2) ==> (+ 5 7)
```

Um dieses Ergebnis zu vereinfachen, können wir statt `(+ 5 7)` den Wert 12 zurückgeben. Die Vereinfachung der gebildeten Summe übernimmt das Konstrukt `Vereinfache-Summe`, das dazu auf Teile der gebildeten Summe zugreifen muss. Für den Konstruktor `Bilde-Summe` benötigen wir korrespondierende Selektoren, damit `Vereinfache-Summe` auf seine Teile zugreifen kann.

Das Konstrukt `Bilde-Summe` gibt eine Liste mit dem Additionsoperator als erstes Element (Präfixnotation) zurück. Damit erzeugt `Bilde--Summe` die gleiche Struktur wie bei einer eingegebenen Summe, die es abzuleiten gilt, d. h. der Wert an den `Formel` gebunden ist. Wir können daher für `Vereinfache-Summe` auf die selben Selektoren zurückgreifen. `Erster-Summand` und `Zweiter-Summand` sind damit (auch) die korrespondierenden Selektoren zu `Bilde-Summe`. Entsprechendes gilt

für den Konstruktor `Bilde-Produkt` mit seinen Selektoren `Multiplikand` und `Multiplikator` in Verbindung mit `Vereinfache-Produkt`.

Hinweis: Selektoraufgabe.
Die Doppelaufgabe der Selektoren `Erster-Summand` und `Zweiter-Summand`, d. h. einerseits den Wert von `Formel` und andererseits den von `Bilde-Summe` erzeugten zu „zerlegen", macht sich bei einer Programmänderung erschwerend bemerkbar. Dies gilt entsprechend auch für die Selektoren `Multiplikand` und `Multiplikator`. Wollen wir z. B. zur Infix-Notation bei der Eingabe (Wert für `Formel`) übergehen und die Präfix-Notation für `Bilde-Summe` beibehalten, dann muss die Doppelfunktion erkannt werden. Jeweils passende Selektoren sind dann neu zu programmieren.

Im Sinne unserer angestrebten Verbundeigenschaft (↪ Abschnitt 2.1.2 S. 146) gilt in dem Konstrukt `Ableiten` z. B.:

```
eval> (Erster-Summand (Bilde-Summe 5 7)) ==> 5
eval> (Zweiter-Summand (Bilde-Summe 5 7)) ==> 7
eval> (Multiplikand (Bilde-Produkt 2 3)) ==> 2
eval> (Multiplikator (Bilde-Produkt 2 3)) ==> 3
```

In der Programmier(alltags)praxis wird diese Eigenschaft einer Konstruktion kaum beachtet und selten realisiert (↪ z. B. die Lösungen zur Ableitungsaufgabe in der oben genannten Literatur). Bezogen auf das Programm `Ableiten` wäre z. B. die folgende Lösung weniger zweckmäßig, wenn als korrespondierende Selektoren `Erster-Summand` und `Zweiter-Summand` beibehalten werden. Sie fasst die Konstruktoren `Vereinfache-Summe` mit `Bilde-Summe` zu einem Konstrukt `Make--Summe` zusammen:

```
eval> (define Make-Summe (lambda (S_1 S_2)
           (cond((and (number? S_1) (number? S_2))
                      (+ S_1 S_2))
                ((and (number? S_1) (zero? S_1))
                     S_2)
                ((and (number? S_2) (zero? S_2))
                     S_1)
                ((Gleiche-Variable? S_1 S_2)
                 (list '* 2 S_1))
                (#t (list '+ S_1 S_2)))))
```

Die Selektion eines Summanden wäre bei diesem Konstruktor nicht mehr in jedem Fall möglich.

```
eval> (Erster-Summand (Make-Summe 5 7))
        ==> ERROR ...  ;Grund: (list-ref 12 1) !
```

`Ableiten`				
○ Ableiten einer Konstanten: $\frac{dc}{dx} = 0$ Linke Seite: P: `Konstante?` → `number?`	○ Ableiten einer Variablen: $\frac{du}{dx} = \begin{cases} 1 & falls \quad u = x \\ 0 & falls \quad sonst \end{cases}$ Linke Seite: P: `Variable?` → `symbol?`; `Gleiche-Variable?` → `eq?`	○ Ableiten einer Summe: $\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx}$ Linke Seite: P: `Summe?` → `pair?`, `eq?`, `car` S: `Erster-Summand`, `Zweiter-Summand` → `car`, `cdr` Rechte Seite: K: `Bilde-Summe` → `list`, `+`, `Vereinfache-Summe` → `+`, `number?`, `Bilde-Produkt` S: ↪ Linke Seite	○ Ableiten eines Produktes: $\frac{d(u*v)}{dx} = (\frac{du}{dx}) * v + u * (\frac{dv}{dx})$ Linke Seite: P: `Produkt?` → `pair?`, `eq?`, `car` S: `Multiplikand`, `Multiplikator` → `car`, `cdr` Rechte Seite: K: `Bilde-Produkt`, `Bilde-Summe` → `list`, `+`, `Vereinfache-Produkt` → `*`, `number?`, `Bilde-Produkt` S: ↪ Linke Seite	○ Sonst: Fehler

Legende: ↪ Programmcode S. 153 — Linke Seite: ≡ Linke Seite der entsprechenden Ableitungsregel; Rechte Seite: ≡ Rechte Seite der entsprechenden Ableitungsregel; P: ≡ Prädikat (Erkennungskonstrukt); S: ≡ Selektor; K: ≡ Konstruktor; →≡ basiert auf, z. B. basiert `Konstante?` auf dem eingebauten `number?`-Konstrukt; ° ≡ Kennzeichnung einer Selektion (↪ S. 80)

Tabelle 2.4: Strukturskizze des Programms `Ableiten`

2.1.4 Zusammenfassung: Abstraktionsebene

Der „Denkrahmen“ des Konstrukteurs, allgemein als Paradigma bezeichnet, wird bestimmt von charakteristischen Konstruktionen, die z. B. imperativ-, funktions- oder objekt-geprägt sind. Ein Paradigma umfasst elementare Konstrukte, Verknüpfungs- und Abstraktionsmittel.

Wir betrachten eine Konstruktion auf den verschiedenen Abstraktionsebenen als einen Verbund von:

1. Konstruktor,
2. Selektor (incl. Präsentator),
3. Prädikat und
4. Mutator (incl. Destruktor).

Von diesem Verbund fordern wir: Die Bausteine, die ein Konstruktor zusammenfügt, müssen von Prädikaten erkannt und durch Selektoren ohne Verlust wiedergewonnen werden können.

Charakteristisches Beispiel für Abschnitt 2.1

`Formular` Hinweis: Zur Nutzung der Mutatoren ↪ S. 164.

```
eval> ;;;;Abbildung eines Formulars
      ;;;Untere Abstraktionsebene:
      (define Bilde-Leeres-Formular
        (lambda ()
          (list '*KOPF* '*TEXT* '*FUSS* '*FORM*)))
eval> ;;;Prädikate
      (define Kopf?
        (lambda (x) (string? x)))
eval> (define Text?
        (lambda (x) (string? x)))
eval> (define Fuss?
        (lambda (x) (string? x)))
eval> (define Leeres-Formular?
        (lambda (x)
          (and (pair? x)
               (eq? (list-ref x 0) '*KOPF*)
               (eq? (list-ref x 1) '*TEXT*)
               (eq? (list-ref x 2) '*FUSS*)
               (eq? (list-ref x 3) '*FORM*))))
eval> ;;;Nächst höhere Abstraktionsebene:
      ;;Konstruktor
      (define Bilde-Formular
        (lambda (Kopf Text Fuss)
          (if (and
               (Kopf? Kopf)
```

```
                (Text? Text)
                (Fuss? Fuss))
               (cons Kopf
                     (cons Text
                           (cons Fuss
                                 (cdddr
                        (Bilde-Leeres-Formular)))))
               (error "Kein Formular gebildet!"))))
eval> ;;Prädikat
      (define Formular?
        (lambda (x)
          (or (Leeres-Formular? x)
              (and (pair? x)
                   (eq? (list-ref x 3) '*FORM*)))))
eval> ;;;Selektoren
      ;;; Name beginnt üblicherweise mit
      ;;; Get-...
      (define Get-Kopf
        (lambda (x)
          (cond ((Formular? x) (list-ref x 0))
                (#t (error "Kein Formular: " x)))))
eval> (define Get-Text
        (lambda (x)
          (cond ((Formular? x) (list-ref x 1))
                (#t (error "Kein Formular: " x)))))
eval (define Get-Fuss
        (lambda (x)
          (cond ((Formular? x) (list-ref x 2))
                (#t (error "Kein Formular: " x)))))
;;;Mutatoren
;;; Hinweis: Das set-car!-Konstrukt wird
;;; später erläutert. Es ist hier
;;; erforderlich, um ein existierendes
;;; Formular, das durch eine Liste
;;; abgebildet ist,
;;; modifizieren zu können.
eval> (define Set-Kopf!
        (lambda (Kopf Form)
          (if (and (Kopf? Kopf)
                   (Formular? Form))
              (begin
                (set-car! Form Kopf)
                Form)
              (error "Fehler!"))))
eval> (define Set-Text!
        (lambda (Text Form)
          (if (and (Text? Text)
                   (Formular? Form))
              (begin
```

```
                (set-car! (cdr Form) Text)
                Form)
              (error "Fehler!"))))
eval> (define Set-Fuss!
        (lambda (Fuss Form)
          (if (and (Fuss? Fuss)
                   (Formular? Form))
              (begin
                (set-car! (cddr Form) Fuss)
                Form)
              (error "Fehler!"))))
;;;;Beispiel
eval> (define F-1 (Bilde-Formular
                    "Leuphana"
                    "Bescheid"
                    "Stand: Entwurf"))
eval> (Get-Text F-1) ==> "Bescheid"
eval> (Get-Fuss (Set-Fuss!
                  "Stand: Genehmigt" F-1))
       ==> "Stand: Genehmigt"
```

2.2 Abbildungsoption: Liste

Für die Abbildungsoption *Liste* haben wir bisher den Konstruktor `cons`, die Selektoren `car` und `cdr` sowie die Prädikate `null?` und `pair?` in unterschiedlichen Zusammenhängen genutzt. Es wurden z. B. neue Listen erzeugt, Elemente am Anfang einer Liste eingefügt, Listen aneinander gekoppelt, die Reihenfolge der Elemente umgedreht oder die Länge einer Liste bestimmt. Wir können die Liste im Sinne einer Signatur (↪ Abschnitt 1.1.4 S. 32) mit diesen Operationen definieren (↪ Abbildung 2.2 S. 161). Dabei unterstellen wir eine endliche, nicht zirkuläre Liste.

Vergleicht man das Konzept der Liste mit dem der Menge, dann sind zwei Aspekte bedeutsam:

1. die Berücksichtigung der Reihenfolge und
2. das mehrfache Vorkommen gleicher Elemente (*Multiplizität*).

Die Liste ist ein Konzept mit Reihenfolge und Multiplizität. Die Menge (engl.: *set*) ist ein Konzept ohne Reihenfolge und ohne Multiplizität. Konzepte ohne Reihenfolge, aber mit Multiplizität werden *bags* oder *multisets* genannt (z. B. in Smalltalk, ↪ [73]).

Für den Selektor lässt sich Multiplizität auch mit Hilfe einer zirkulären Liste simulieren. Bei ihr ist der Nachfolger des (quasi) letzten Elementes das erste Element der Liste („Die Katze beißt sich in den Schwanz!“). Wir nutzen die zirkuläre Liste im Beispiel „Verwalten von

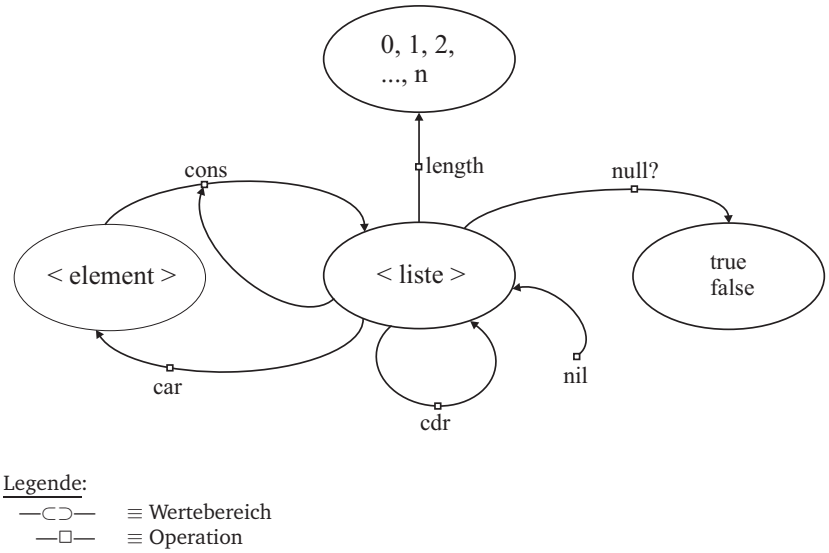


Abbildung 2.2: Signatur einer endlichen, nicht zirkulären Liste

Arbeitsvorgängen“ (↪ Abschnitt 2.2.1 S. 166). Dabei unterstellen wir einen Manager, der für größere Arbeiten eine Bearbeitsungsreihenfolge festlegt. Zur Wiederholung der bisher verwendeten Listen-Konstrukte zeigen wir die Lösung zunächst ohne Listen-Mutatoren. Mit destruktiven Listenveränderungen definieren wir anschließend die Lösung, abgebildet als zirkuläre Liste.

Zur Konstruktion einer zirkulären Liste wird eine schon existierende Liste modifiziert. Zum leichteren Verstehen dieser Listen-Mutatoren befassen wir uns mit `cons`-Zellen-Skizzen (↪ Abschnitt 1.2.1 S. 52, Abbildung 1.15 S. 60). Mit diesen `cons`-Zellen-Strukturen sind keinesfalls alle realisierten Implementationen einer Liste beschrieben (↪ z. B. LISP-Maschinen ↪ Abschnitt A.1 S. 460). Eine übliche Abweichung ist die Reihenfolge von `car`- und `cdr`-Teil. Weil die Applikation des `cdr`-Konstruktes häufiger vorkommt als die des `car`-Konstruktes, stellen z. B. in einer `cons`-Zelle von 8 Byte Länge die ersten vier Bytes den `cdr`-Teil dar.

Die Assoziationsliste (kurz: A-Liste) behandeln wir anhand der Definition eines Kontos (↪ Abschnitt 2.2.2 S. 183). Die Eigenschaftsliste (engl.: *property list*; kurz: P-Liste) wird mit dem Beispiel „Laufmappen-Transportsteuerung“ erläutert. In einem Bürogebäude ist für die Steuerung eines mechanischen Fließbandsystems ein Weg zwischen Start- und Ziel-Schreibtisch zu ermitteln. Dazu programmieren wir die beiden Suchalgorithmen *Depth First* und $A^\star$ (↪ Abschnitt 2.2.3 S. 191).

Beispiel: Verwalten von Arbeitsvorgängen

Ein Manager bearbeitet seine größeren Aufgaben nach einem Zyklusverfahren. Abhängig vom aktuellen Tagesgeschehen widmet er sich eine Zeit lang einem Vorgang, dann geht er zum nächsten Vorgang über. Hat er alle Vorgänge bearbeitet, fängt er wieder von vorn an. Fertige Vorgänge werden aus diesem Bearbeitungszyklus entfernt. Neue Vorgänge werden fallabhängig in den Zyklus eingeordnet. Zu einem Zeitpunkt t_0 hat er z. B. folgende Vorgänge zu bearbeiten:

1. „Über die Investition Lagerhalle West entscheiden“,
2. „Zeugnis für den Abteilungsleiter Meier diktieren“ und
3. „Budgetplanung für die Zentralabteilung aufstellen“.

Die globale Variable `*ARBEITSPLAN*` stellt diese Vorgänge und ihre Reihenfolge durch eine Liste dar:

```
eval> (define *ARBEITSPLAN*
(list
 "Über die Investition Lagerhalle West entscheiden"
 "Zeugnis für den Abteilungsleiter Meier diktieren"
 "Budgetplanung für die Zentralabteilung aufstellen")
)
```

Der Bearbeitungszyklus könnte auf der Grundlage dieser einfach strukturierten Liste, wie folgt, realisiert werden:

```
eval> (define Vorgangsverwaltung
        (lambda (x)
          (letrec
              ((Anforderung (lambda ()
                              (begin (newline)
      (print "Zeige naechsten Vorgang? J/N:"))
                                     (read)))
               (Zyklus (lambda (Liste)
                         (let ((Antwort (Anforderung)))
                           (cond ((eq? Antwort 'J)
                                  (cond
                                    ;;Umschalten auf
                                    ;; den Anfang der
                                    ;; Liste
                                    ((null? Liste)
                                   (print
                                     (car
                                       *ARBEITSPLAN*))
                                     (Zyklus
                                      (cdr
                                        *ARBEITSPLAN*
                                        )))
```

```
                                        ;;Übergang auf
                                        ;; das nächste
                                        ;; Element
                                        (#t (print
                                             (car Liste))
                                            (Zyklus
                                             (cdr
                                               Liste)))))
                                    (#t "Ende"))))))
            (cond ((null? *ARBEITSPLAN*)
                      "Keine Aufgaben")
                  (#t (Zyklus *ARBEITSPLAN*))))))
eval> (Vorgangsverwaltung *ARBEITSPLAN*)
      ==>
"Zeige naechsten Vorgang? J/N:"J
"Über die Investition Lagerhalle West entscheiden"
"Zeige naechsten Vorgang? J/N:"J
"Zeugnis für den Abteilungsleiter Meier diktieren"
"Zeige naechsten Vorgang? J/N:"J
"Budgetplanung für die Zentralabteilung aufstellen"
"Zeige naechsten Vorgang? J/N:"J
"Über die Investition Lagerhalle West entscheiden"
"Zeige naechsten Vorgang? J/N:"J
"Zeugnis für den Abteilungsleiter Meier diktieren"
"Zeige naechsten Vorgang? J/N:"J
"Budgetplanung für die Zentralabteilung aufstellen"
"Zeige naechsten Vorgang? J/N:"N
"Ende"
```

Das Hinzufügen eines neuen Vorganges oder das Herausnehmen eines fertigen Vorganges ermöglicht der Mutator `set!` in Verbindung mit dem Konstruktor `cons` und dem Selektor `cdr`. Wir können als entsprechende Beispiele definieren:

```
eval> ;;Vorgang hinzufügen
     (set! *ARBEITSPLAN*
           (cons "Urlaub festlegen"
                 *ARBEITSPLAN*))
eval> *ARBEITSPLAN* ==>
("Urlaub festlegen"
 "Über die Investition Lagerhalle West entscheiden"
 "Zeugnis für den Abteilungsleiter Meier diktieren"
 "Budgetplanung für die Zentralabteilung aufstellen")

eval> ;;Ersten Vorgang löschen
      (set! *ARBEITSPLAN* (cdr *ARBEITSPLAN*))
eval> *ARBEITSPLAN* ==>
("Über die Investition Lagerhalle West entscheiden"
 "Zeugnis für den Abteilungsleiter Meier diktieren"
 "Budgetplanung für die Zentralabteilung aufstellen")
```

Die Veränderung des Symbolwertes leistet das `set!`-Konstrukt, weil es in der jeweiligen Umgebung das Symbol an den neuen Wert bindet. Wir können das `set!`-Konstrukt auch als Mutator des Zeigers `*ARBEITSPLAN*` auffassen, der auf die erste `cons`-Zelle der Liste zeigt (↪ Abbildung 2.3 S. 165).

Mit dem `set!`-Konstrukt kann ein Zeiger, der auf eine `cons`-Zelle zeigt, auf eine andere vorhandene `cons`-Zelle gesetzt werden. Die Zeiger des `car`-Parts bzw. des `cdr`-Parts einer `cons`-Zelle bleiben davon unberührt. Für ihre Änderung gibt es eigene Mutatoren, die Konstrukte `set-car!` und `set-cdr!`.

Mutatoren für die `cons`-Zelle

`eval>` `(set-car!` $<cons-zelle><sexpr>$`)`
Nebeneffekt: $<modifizierte-zelle>$
`eval>` `(set-cdr!` $<cons-zelle><sexpr>$`)`
Nebeneffekt: $<modifizierte-zelle>$
mit:

$<cons-zelle>\equiv$	Symbolischer Ausdruck, der eine `cons`-Zelle als Wert hat.
$<modifizierte-zelle>\equiv$	`cons`-Zelle mit dem Wert von $<sexpr>$ im `car`- oder `cdr`-Teil

Die beiden Mutatoren `set-car!` und `set-cdr!` haben die Aufgabe, dem `car`- bzw. `cdr`-Teil einer existierenden `cons`-Zelle einen neuen Wert zu geben. Dazu werten sie ihre Argumente aus. Da wir sie nur für diesen Nebeneffekt verwenden (sollten!), ist ihr eigener Wert prinzipiell nicht spezifiziert.[4]

Hinweis: `set-car!`- und `set-cdr!`-Konstrukt in Scheme.
Damit diese Konstrukte in *PLT-Scheme* (DrScheme) zur Verfügung stehen, ist die die Bibliothek für *Mutable Pairs* von Scheme R6RS vorab zu laden[5] oder die Sprache `R5RS` zu wählen. Wird mit R6RS gearbeitet werden Konstrukte in geschweiften Klammern ausgegegeben.[6]

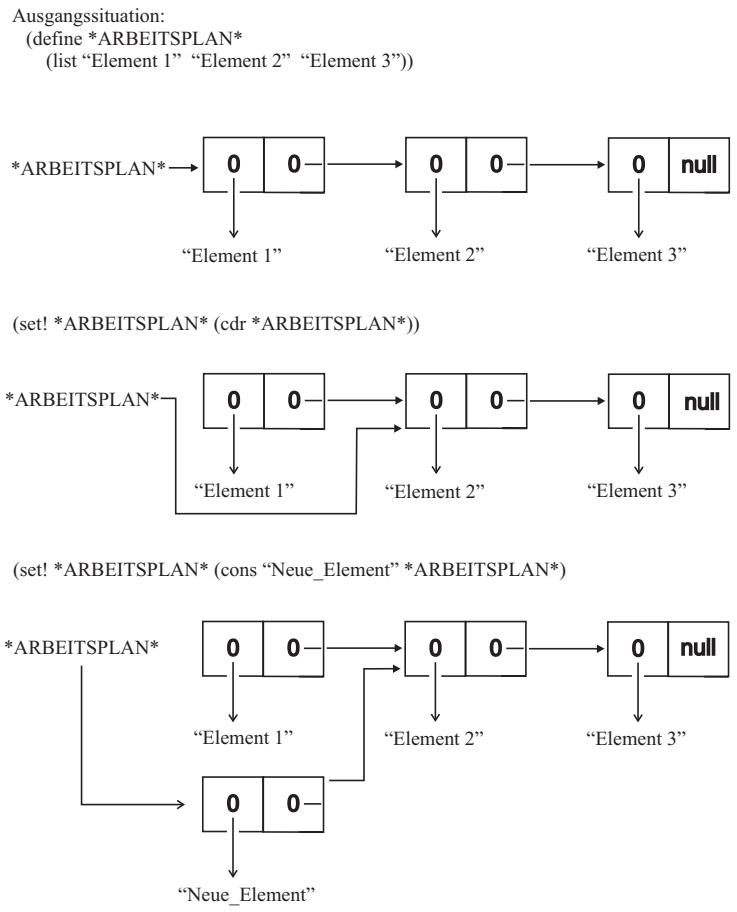
```
eval> (require rnrs/base-6)
eval> (require rnrs/mutable-pairs-6)
```

In vielen LISP-Implementationen, so z. B. in *Common LISP*, haben diese Mutatoren die klassischen Namen: `rplaca` (Akronym für *RePLAce the CAr of* ...) und `rplacd` (Akronym für *RePLAce the CDr of* ...).

[4] In *PLT-Scheme (DrScheme)* ist der Wert `void`. Daher gilt z. B. `(void? (set-car! Foo 7)) ==> #t`.

[5] Zum `require`-Konstrukt ↪ Abschnitt 2.7.1 S. 320.

[6] Geschweifte Klammern sind den runden Klammern gleichzusetzen; ebenso die eckigen Klammern (↪ S. 6).



Legende:

`car`-Part | `cdr`-Part	≡ `cons`-Zelle
∘→	≡ Zeiger (Adresse)
`null`	≡ `null`-Zeiger, entspricht ∘→ `null`

Abbildung 2.3: `set!`-Konstrukt — Änderung des Zeigers auf eine existierende `cons`-Zelle

Beispiele für `set-car!` **und** `set-cdr!`

```
eval> (define Foo '(a b c d))
eval> (set-car! Foo (* 2 3))
eval> (void? (set-cdr! Foo '(x y)))
      ==> #t
eval> Foo ==> {6 x y}
eval> (set-cdr! Foo (* 3 3))
eval> Foo ==> {6 . 9}
```

Die Mutatoren `set-car!` und `set-cdr!` können Transparenz-Probleme bewirken. Nicht leicht durchschaubare Nebeneffekte sind formulierbar, wie das folgende Beispiel zeigt:

```
eval> (define Foo
        (cons (cons #t #t) (cons #t #t)))
eval> Foo ==> ((#t . #t) #t . #t)
eval> (define Bar
        (let ((x (cons #t #t)))
          (cons x x)))
eval> Bar ==> ((#t . #t) #t . #t)
eval> (equal? Foo Bar) ==> #t
eval> (set-cdr! (cdr Foo) 'Neu)
eval> (set-cdr! (cdr Bar) 'Neu)
```

Nachdem sowohl auf `Foo` wie auf `Bar` die gleiche `set-cdr!`-Formulierung angewendet wurde, erwartet man spontan, dass „Gleiches“ weiterhin gleich ist, d. h. `(equal? Foo Bar)` weiterhin den Wert `true` hat. Dies ist jedoch nicht der Fall!

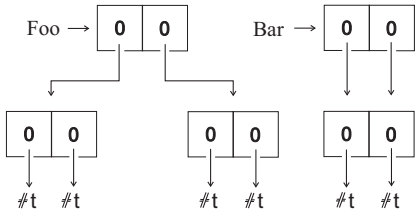
```
eval> (equal? Foo Bar) ==> #f
eval> Foo ==> ((#t . #t) #t . neu)
eval> Bar ==> ((#t . neu) #t . neu)
```

Dieser Nebeneffekt leuchtet unmittelbar ein, wenn man das `cons`-Zellen-Modell für `Foo` und `Bar` betrachtet. Die Abbildung 2.4 S. 167 zeigt dieses vor der Applikation des Mutators `set-cdr!`.

Problematisch sind Mutatoren immer dann, wenn ihre „Zerstörung“ sich auf mehrfach genutzte „Objekte“ bezieht. Als Merksatz formuliert: Sind Mutatoren dabei, dann ist „Gleiches durch Gleiches“ nicht einfach ersetzbar. (Näheres zur Semantik destruktiver Konstrukte ↪ z. B. [120]).

2.2.1 Zirkuläre Liste

Im obigen Beispiel „Verwalten von Arbeitsvorgängen“ können wir mit dem `set-cdr!`-Konstrukt den Bearbeitungszyklus einfach dadurch abbilden, dass der Zeiger des `cdr`-Parts der letzten `cons`-Zelle von `*ARBEITSPLAN*` auf die erste `cons`-Zelle zeigt. Nehmen wir an, dass die Liste `*ARBEITSPLAN*` drei Elemente enthält, dann wäre formulierbar:



Legende:
↪ Abbilung 2.3 S. 165.

Abbildung 2.4: (equal? Foo Bar) ==> #t

```
eval> (define *ARBEITSPLAN*
(list
 "Über die Investition Lagerhalle West entscheiden"
 "Zeugnis für den Abteilungsleiter Meier diktieren"
 "Budgetplanung für die Zentralabteilung aufstellen")
)
eval> (set-cdr! (cddr *ARBEITSPLAN*) *ARBEITSPLAN*)
```

Aufgrund des *READ-EVAL-PRINT*-Zyklus wird die Liste `*ARBEITSPLAN*` auf dem Bildschirm ausgegeben. `*ARBEITSPLAN*` stellt jetzt jedoch eine zirkuläre Liste (*infinite Sequenz*) dar, weil der letzte `cdr`-Part-Zeiger wieder auf die Anfangs-`cons`-Zelle zeigt. Zur Vermeidung einer endlosen Ausgabe gibt es folgende Darstellung:

```
eval> *ARBEITSPLAN*
#0=
("Über die Investition Lagerhalle West entscheiden"
 "Zeugnis für den Abteilungsleiter Meier diktieren"
 "Budgetplanung für die Zentralabteilung aufstellen"
 .
 #0#)
```

Dass `*ARBEITSPLAN*` an eine zirkuläre Liste gebunden ist, zeigt die Anwendung des `eq?`-Konstruktes.

```
eval> (eq? *ARBEITSPLAN* (cdddr *ARBEITSPLAN*))
      ==> #t
```

Das `eq?`-Konstrukt stellt fest, ob die ihm übergebenen Zeiger auf dieselbe `cons`-Zelle zeigen, d. h. denselben Wert haben. Die Identitätsentscheidung des `eq?`-Konstruktes (↪ Abschnitt 1.2.1 S. 59) ist das Prüfen von Zeigern.

Um die Elemente einer zirkulären Liste auszugeben, definierten wir ein eigenes `Print-Element`-Konstrukt. Mit Hilfe des `read`-Konstruktes können wir Daten vom Benutzer abfragen, so dass zwischen weitere Ausgabe und Ende untschieden werden kann. Die Ausgabe selbst veranlasst das `display`-Konstrukt. Eine neue Zeile bei der Ausgabe erreichen wir mit dem `newline`-Konstrukt.

```
eval> (define Print-Element
        (lambda (Zirkulaere_Liste)
          (display (car Zirkulaere_Liste))
          (newline)
                 (let ((Anforderung
                        (begin
                          (newline)
                  (display
                    "Nächstes Element? J/N : ")
                          (read))))
                   (cond ((eq? Anforderung 'J)
                          (Print-Element
                            (cdr Zirkulaere_Liste)))
                         (#t "Ende!")))))

eval> (Print-Element *ARBEITSPLAN*)
      ==>
Über die Investition Lagerhalle West entscheiden

Nächstes Element? J/N : J
Zeugnis für den Abteilungsleiter Meier diktieren

Nächstes Element? J/N : J
Budgetplanung für die Zentralabteilung aufstellen

Nächstes Element? J/N : J
Über die Investition Lagerhalle West entscheiden

Nächstes Element? J/N : N
"Ende!"
```

Mit Hilfe der Mutatoren `set!`, `set-car!` und `set-cdr!` ist die Verwaltung der Arbeitsvorgänge auf der Grundlage des Konzepts der zirkulären Liste im Programm (↪ Programmcode S. 170) abgebildet. Dieses Programm hat folgende Struktur:

```
eval> (define Verwalten-von-Arbeitsvorgaengen!
(lambda (Z_Liste)
  (letrec
     ((Anforderung ... (read))
       ;Hauptfunktion
      (Vorgangsverwaltung
       (lambda (Zirkulaere_Liste)
```

```
        ;Steuerungscodeabfrage
       (let ((Antwort (Anforderung)))
         (cond
               ;Nächster Vorgang
               ((eq? Antwort 'Ja) ... )
               ;Neuer Vorgang
               ((eq? Antwort 'Neu) ...)
               ;Lösche Vorgang
               ((eq? Antwort 'OK) ...)
               ;Programm beenden
               ((eq? Antwort 'Stop)...)
               ;Vorgang anzeigen
               (#t ... ))))))
    ;Aufruf der lokalen
    ; end-rekursiven Hauptfunktion
   (Vorgangsverwaltung Z_Liste))))
```

Zum Verstehen der Aufgabe sind vorab Anforderungen und Testfälle formuliert. Dabei verweist die Abkürzung `A` auf eine *Anforderung* und die Abkürzung `T` auf einen *Testfall*.

`A1`: `Verwalten-von-Arbeitsvorgaengen!` speichert Vorgänge in einer zirkulären Liste.

- `A1.1`: Die Vorgänge können in der gespeicherten Reihenfolge nacheinander angezeigt werden.
- `A1.2`: Der angezeigte Vorgang ist aus der Liste entfernbar.

`A2`: Ein neuer Vorgang ist in die Liste einfügbar:

- `A2.1`: Vor dem gezeigten Vorgang oder
- `A2.2`: hinter dem gezeigten Vorgang.

`T1`: Beispiel

- `T1.1`: Arbeitsplan:

```
eval> (define *ARBEITSPLAN*
(list
"Über die Investition Lagerhalle West entscheiden"
"Zeugnis fuer den Abteilungsleiter Meier diktieren"
"Budgetplanung fuer die Zentralabteilung aufstellen"
))
```

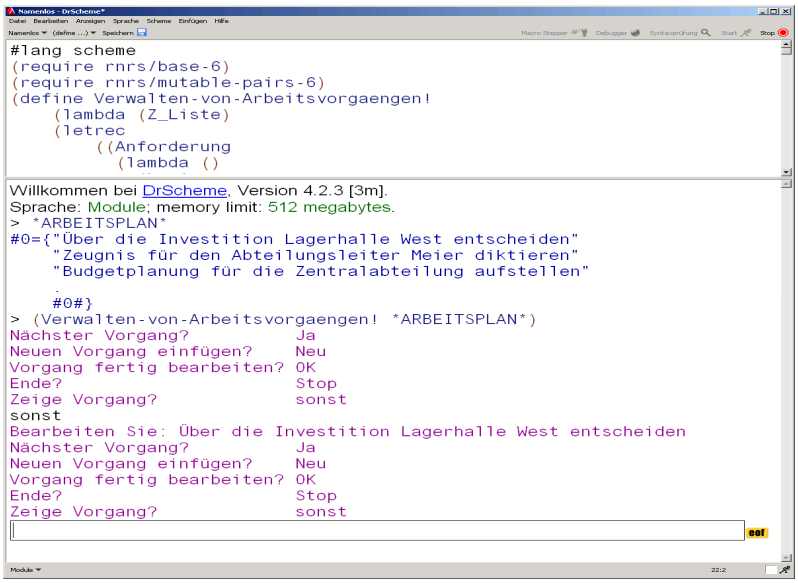
- `T1.2`: Applikation:

```
eval> (Verwalten-von-Arbeitsvorgaengen!
        *ARBEITSPLAN*)
      ==> ... ;Geführte Benutzereingaben
```

Programm: `Verwalten-von-Arbeitsvorgaengen!`

```
;;;Scheme R6RS mit Mutable Pairs
eval> (require rnrs/base-6)
eval> (require rnrs/mutable-pairs-6)

eval> (define Verwalten-von-Arbeitsvorgaengen!
    (lambda (Z_Liste)
    (letrec
        ((Anforderung
          (lambda ()
            (begin
              (display "Nächster Vorgang?          Ja")
              (newline)
              (display "Neuen Vorgang einfügen?    Neu")
              (newline)
              (display "Vorgang fertig bearbeiten? OK")
              (newline)
              (display "Ende?                      Stop")
              (newline)
              (display "Zeige Vorgang?             sonst")
              (newline)
              (read))))
         ;;Hauptfunktion
         (Vorgangsverwaltung
          (lambda (Zirkulaere_Liste)
            (let
                ((Antwort (Anforderung))) ;Steuerungs-
                                          ; codeabfrage
              (cond
                ;;Zum nächsten Vorgang
                ((eq? Antwort 'Ja)
                 (Vorgangsverwaltung
                   (cdr Zirkulaere_Liste)))
                ;;Neuen Vorgang einbauen
                ((eq? Antwort 'Neu)
                 (let ((Neuer-Vorgang
                        (begin
                          (display "Bitte geben Sie ")
                          (newline)
        (display "Ihren Vorgang ein! (Zeichenkette): ")
                          (newline)
                          (read)))
                       (Neue-cons-Zelle
                        (list "Platzhalter")))
                   (cond
                     ((eq? (car Zirkulaere_Liste) (list))
                      (set-car!
                        Zirkulaere_Liste Neuer-Vorgang)
                      (Vorgangsverwaltung Zirkulaere_Liste))
                     ;;Vor dem aktuellen Vorgang den neuen
                     ;; Vorgang einbauen? ;vgl. A2.1
                     ((eq? 'Ja
                           (begin
                             (display
```

```
 "Soll Ihr neuer Vorgang ")
             (display
 "vorher eingefügt werden? Ja")
             (newline)
             (read)))
      (set-cdr! Neue-cons-Zelle
                (cdr Zirkulaere_Liste))
      (set-car! Neue-cons-Zelle
                (car Zirkulaere_Liste))
      (set-car! Zirkulaere_Liste
                Neuer-Vorgang)
      (set-cdr! Zirkulaere_Liste
                Neue-cons-Zelle)
      (Vorgangsverwaltung
        Zirkulaere_Liste))
     ;;vgl. A2.2
     (#t (set-cdr!
          Neue-cons-Zelle
          (cdr Zirkulaere_Liste))
         (set-car! Neue-cons-Zelle
                   Neuer-Vorgang)
         (set-cdr! Zirkulaere_Liste
                   Neue-cons-Zelle)
         (Vorgangsverwaltung
          Zirkulaere_Liste)))))
;;;Vorgang aus Liste entfernen
;;;vgl. A1.2
((eq? Antwort 'OK)
 (letrec
     ((Quasi-letzte-Element
       (lambda (Listen_Zeiger
                Aktuelle_Zeiger
                Vorhergehende_Position)
         (cond
           ((eq? Listen_Zeiger
                 Aktuelle_Zeiger)
            Vorhergehende_Position)
           (#t (Quasi-letzte-Element
                Listen_Zeiger
                (cdr Aktuelle_Zeiger)
                (cdr
                  Vorhergehende_Position
                )))
           ))))
   (set-cdr!
    (Quasi-letzte-Element
     Zirkulaere_Liste
     (cddr Zirkulaere_Liste)
     (cdr Zirkulaere_Liste))
    (cdr Zirkulaere_Liste))
   (set-car! Zirkulaere_Liste
             (list))
   (set! Zirkulaere_Liste
         (cdr Zirkulaere_Liste))
   (Vorgangsverwaltung
    Zirkulaere_Liste)))
```



Legende:
Dr. Scheme ↪http://www.plt-scheme.org/ (Zugriff: 24-Oct-2009) — Quellcode S. 170.

Abbildung 2.5: Applikation: `Verwalten-von-Arbeitsvorgaengen!`

```
                ;;;Programm beenden
                ((eq? Antwort 'Stop)
                 (display
               "Vorgangsverwaltung beendet."))
                ;;;Vorgang anzeigen vgl. A1.1
                (#t (display "Bearbeiten Sie: ")
                    (display
                      (car Zirkulaere_Liste))
                    (newline)
                    (Vorgangsverwaltung
                     Zirkulaere_Liste)))))))
      ;;Start der lokalen
      ;; endrekursiven Hauptfunktion
      (Vorgangsverwaltung Z_Liste))))
```

Hier das Protokoll einer beispielhaften Applikation von `Verwalten-von-Arbeitsvorgaengen!` (↪ auch Abbildung 2.5 S. 172):

```
eval>  (define *ARBEITSPLAN* (list
 "Über die Investition Lagerhalle West entscheiden"
 "Zeugnis für den Abteilungsleiter Meier diktieren"
```

```
 "Budgetplanung für die Zentralabteilung aufstellen"))
eval> (set-cdr! (cddr *ARBEITSPLAN*) *ARBEITSPLAN*)
eval> *ARBEITSPLAN* ==>
#0={"Über die Investition Lagerhalle West entscheiden"
    "Zeugnis für den Abteilungsleiter Meier diktieren"
    "Budgetplanung für die Zentralabteilung aufstellen"
    .
    #0#}
eval> (Verwalten-von-Arbeitsvorgaengen! *ARBEITSPLAN*)
      ==>
Nächster Vorgang?          Ja
Neuen Vorgang einfügen?    Neu
Vorgang fertig bearbeiten? OK
Ende?                      Stop
Zeige Vorgang?             sonst
sonst
Bearbeiten Sie: Über die Investition Lagerhalle West entscheiden
Nächster Vorgang?          Ja
Neuen Vorgang einfügen?    Neu
Vorgang fertig bearbeiten? OK
Ende?                      Stop
Zeige Vorgang?             sonst
Ja
Nächster Vorgang?          Ja
Neuen Vorgang einfügen?    Neu
Vorgang fertig bearbeiten? OK
Ende?                      Stop
Zeige Vorgang?             sonst
sonst
Bearbeiten Sie: Zeugnis für den Abteilungsleiter Meier diktieren
Nächster Vorgang?          Ja
Neuen Vorgang einfügen?    Neu
Vorgang fertig bearbeiten? OK
Ende?                      Stop
Zeige Vorgang?             sonst
Neu
Bitte geben Sie
Ihren Vorgang ein! (Zeichenkette):
"Mietvertrag erneuern"
Soll Ihr neuer Vorgang vorher eingefügt werden? Ja
Ja
Nächster Vorgang?          Ja
Neuen Vorgang einfügen?    Neu
Vorgang fertig bearbeiten? OK
Ende?                      Stop
Zeige Vorgang?             sonst
sonst
Bearbeiten Sie: Mietvertrag erneuern
Nächster Vorgang?          Ja
Neuen Vorgang einfügen?    Neu
Vorgang fertig bearbeiten? OK
Ende?                      Stop
Zeige Vorgang?             sonst
stop
Vorgangsverwaltung beendet.
eval> *ARBEITSPLAN* ==>
```

```
#0={"Über die Investition Lagerhalle West entscheiden"
    "Mietvertrag erneuern"
    "Zeugnis für den Abteilungsleiter Meier diktieren"
    "Budgetplanung für die Zentralabteilung aufstellen"
    .
    #0#}
```

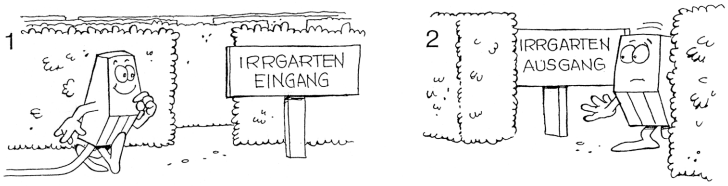
Erläuterung des Programmes: `Verwalten-von-Arbeitsvorgaengen!`

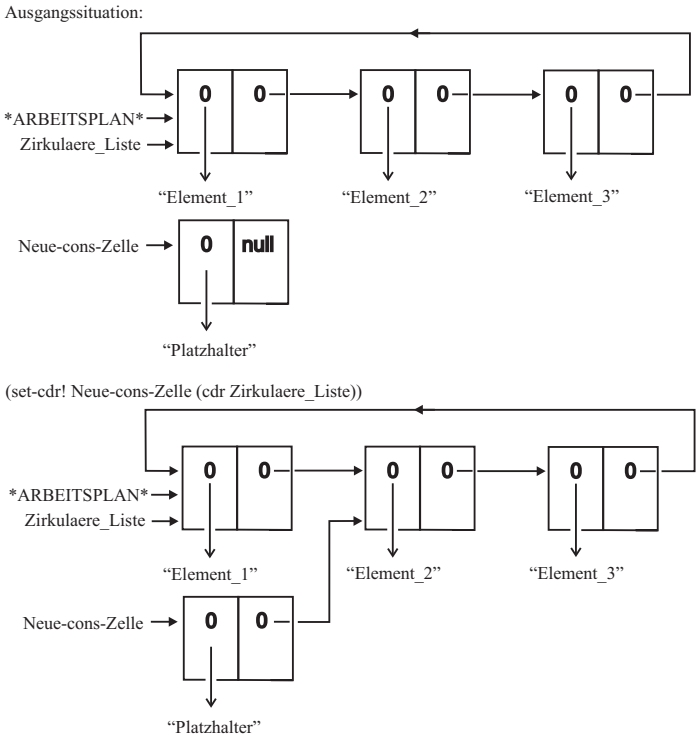
Zur Erläuterung wird angenommen, dass beim Aufruf des Programmes (↪ Programmcode S. 170) das Argument `*ARBEITSPLAN*` an eine zirkuläre Liste mit drei Elementen gebunden ist. Damit ist zunächst die `lambda`-Variable `Z_Liste` an die zirkuläre Liste von `*ARBEITSPLAN*` gebunden. Über die Applikation von `(Vorgangsverwaltung Z_Liste)` wird die `lambda`-Variable von `Vorgangsverwaltung`, also `Zirkulaere_Liste`, ebenfalls an diese zirkuläre Liste gebunden. Anders formuliert: Der Zeiger `Zirkulaere_Liste` weist auf dieselbe `cons`-Zelle wie der Zeiger `*ARBEITSPLAN*`.

Unterstellen wir nun, dass der Benutzer mit der Eingabe `Neu` antwortet und seinen neuen Vorgang am „Anfang“ der zirkulären Liste einfügen will, dann zeigen die Abbildungen 2.6 S. 175 und 2.7 S. 176 die Änderung der Zeiger auf die `cons`-Zellen. Nochmals sei darauf hingewiesen, dass es sich nur um ein grafisches Modell für das Verstehen der Mutatoren `set!`, `set-car!` und `set-cdr` handelt.

In den Abbildungen 2.6 S. 175 und 2.7 S. 176 werden mit der Anwendung der Mutatoren `set-car!` und `set-cdr!` auf `Zirkulaere_Liste` die Inhalte derjenigen `cons`-Zellen geändert, auf die sowohl die `lambda`-Variable `Zirkulaere_Liste` selbst als auch die globale Variable `*ARBEITSPLAN*` zeigen. Nach Beendigung des Programms wird die Bindung der `lambda`-Variable wieder aufgehoben. Die Bindung der globalen Variable `*ARBEITSPLAN*` bleibt bestehen, d. h. es gibt weiterhin in der *top-level*-Umgebung die Assoziation zwischen dem Symbol `*ARBEITSPLAN*` und den modifizierten `cons`-Zellen.

Die Abbildung 2.8 S. 177 zeigt das Einfügen eines neuen Vorganges nach dem aktuellen Vorgang. Hier ist wieder das Aufbrechen einer `cons`-Zellen-Kette und das Einfügen einer zusätzlichen `cons`-Zelle durch den Mutator `set-cdr!` abgebildet.



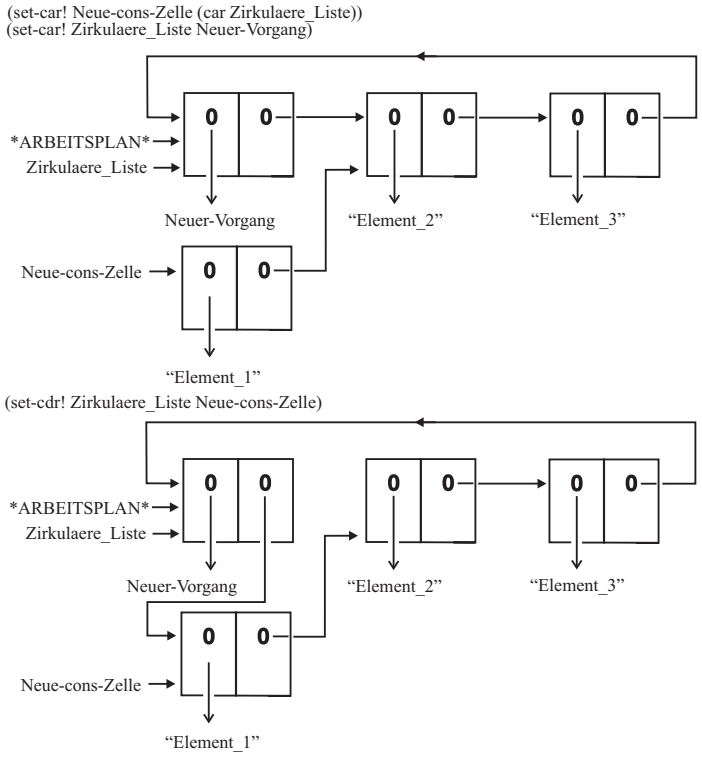


Legende:
↪ Symbole S. 165; ↪ Programmcode S. 170, ↪ Teil II S. 176

Abbildung 2.6: Neuen Vorgang am „Anfang“ der zirkulären Liste einfügen `cons`-Zelle — Teil I

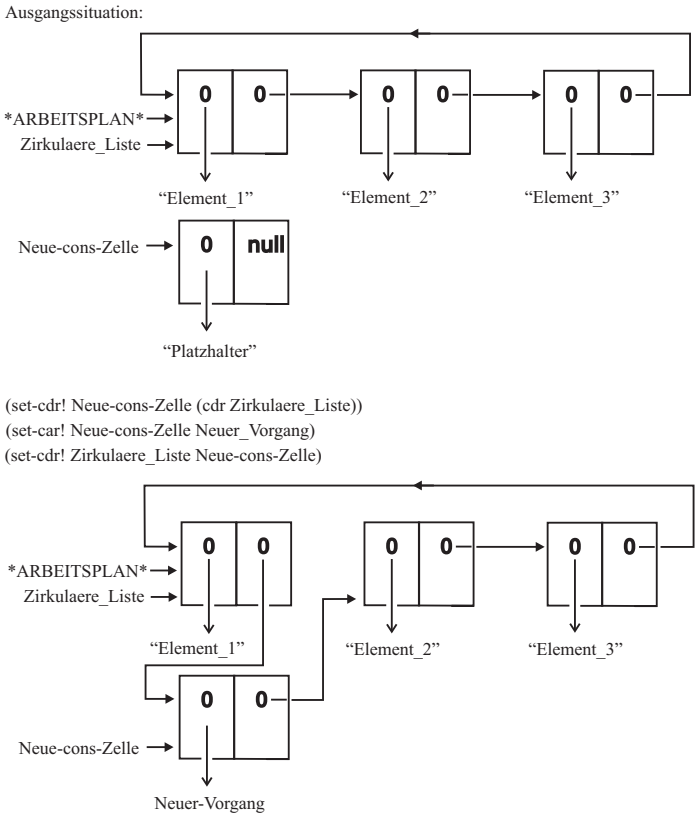
Bei der zirkulären Liste können wir mit dem `cdr`-Konstrukt auf die nachfolgende `cons`-Zelle positionieren. In Pfeilrichtung lässt sich der Zirkel durchlaufen, in der entgegengesetzten Richtung nicht. Soll der `cdr`-Part einer vorgehenden `cons`-Zelle modifiziert werden, z. B. um die aktuelle `cons`-Zelle zu überspringen, dann müssen wir eine Hilfslösung wählen. Wir durchlaufen den Zyklus in der `cdr`-Richtung so weit, bis wir an die vorhergehende `cons`-Zelle kommen. Diese Hilfslösung bildet das Konstrukt `Quasi-letzte-Element` ab (↪ Abbildung 2.9 S. 178).

Das `Quasi-letzte-Element`-Konstrukt gibt als Wert einen Zeiger auf die `cons`-Zelle wieder, die der aktuellen `cons`-Zelle in der zirkulären Liste vorhergeht. Zum Herausnehmen einer `cons`-Zelle aus der zirkulären Liste werden mit den Mutatoren `set-car!` und `set-cdr!` die Inhalte der gemeinsamen `cons`-Zellen von `*ARBEITSPLAN*` und



Legende:
↪ Symbole S. 165; ↪ Programmcode S. 170, ↪ Teil I S. 175

Abbildung 2.7: Neuen Vorgang am „Anfang“ der zirkulären Liste einfügen `cons`-Zelle — Teil II

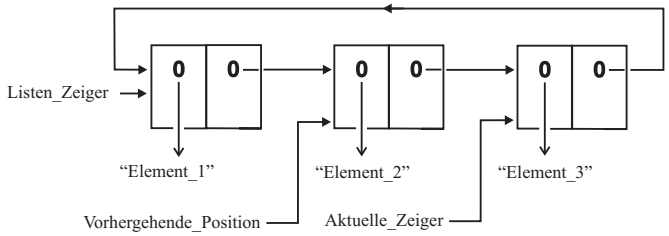


Legende:
↪ Symbole S. 165; ↪ Programmcode S. 170

Abbildung 2.8: Neuen Vorgang in die zirkuläre Liste einfügen

Funktion Quasi-letzte-Element:

```
(letrec
   (( Quasi-letzte-Element
      (lambda (Listen_Zeiger Aktuelle_Zeiger
               Vorhergehende_Position)
               ;;  Das quasi letzte Element der zirkulären Liste wird
               ;;  erkannt durch Vergleich mittels  eq?  wobei
               ;;  Aktuelle_Zeiger und Vorhergehende_Position
               ;;  "synchron" die Liste entlangwandern.
        (cond ((eq? Listen_Zeiger Aktuelle_Zeiger)
               Vorhergehende_Position)
              (#t (Quasi-letzte-Element
                     Listen_Zeiger
                     (cdr Aktuelle_Zeiger)
                     (cdr Vorhergehende_Position)))))))
                        ... )
```



Legende:
↪ Symbole S. 165; ↪ Programmcode S. 170

Abbildung 2.9: Hilfslösung um die quasi letzte `cons`-Zelle einer zirkulären Liste zu ermitteln

Zirkulaere_Liste modifiziert (↪ Abbildung 2.10 S. 180). Mit dem Mutator set!, angewendet auf die lambda-Variable Zirkulaere_Liste, wird jedoch nur dieser Zeiger geändert, d. h. auf die um das erste Element verkürzte zirkuläre Liste gesetzt. Der Zeiger *ARBEITSPLAN* bleibt davon unberührt.

Wird nach dem Löschen eines Vorganges das Programm Verwalten-von-Arbeitsvorgaengen! mit dem Argument *ARBEITSPLAN* ein zweites Mal gestartet, dann wäre von *ARBEITSPLAN* das erste Element null (↪ Abbildung 2.10 S. 180). Um diesen Effekt zu vermeiden, muss der Zeiger *ARBEITSPLAN* modifiziert werden. Dies kann mit dem Mutator set! in der (Definitions-)Umgebung von *ARBEITSPLAN* erfolgen. Dazu könnte z. B. das Programm Verwalten-von-Arbeitsvorgaengen! an der Stelle der Programm-Ende-Behandlung, wie folgt, geändert werden:

bisher:

```
((eq? Antwort 'Stop)
 (display "Vorgangsverwaltung beendet."))
```

Änderung:

```
((eq? Antwort 'Stop)
 (display "Vorgangsverwaltung beendet.")
 Zirkulaere_Liste)
```

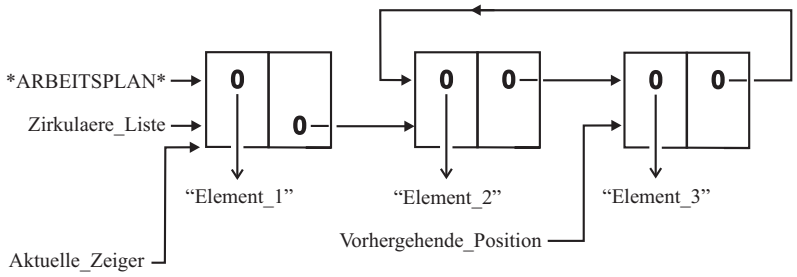
Der Rückgabewert des Programmes Verwalten-von-Arbeitsvorgaengen! ist dann stets der Zeiger Zirkulaere_Liste, so dass dann formuliert werden könnte:

```
eval> (begin
        (set! *ARBEITSPLAN*
          (Verwalten-von-Arbeitsvorgaengen!
            *ARBEITSPLAN*))
       #t) ==>#t
```

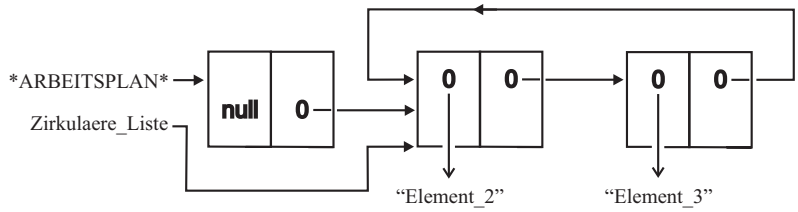
Hinweis: Makro-Nutzung
Einerseits ist diese umständliche Formulierung mit der zweimaligen Nennung von *ARBEITSPLAN* durch die Definition eines Makros syntaktisch vereinfachbar. Andererseits könnte *ARBEITSPLAN* direkt und nicht über die lambda-Schnittstelle des Konstruktes Verwalten-von-Arbeitsvorgaengen modifiziert werden, ↪ z. B. durch die Konstruktion des Programms Aktenverwaltung, Programmcode S. 229.

Wie die Abbildungen 2.6 S. 175 bis 2.10 S. 180 dokumentieren, verändert das Konstrukt Verwalten-von-Arbeitsvorgaengen die cons-Zellen seines Argumentes. Hier ist gerade diese Modifikation der gewünschte Effekt. In anderen Fällen allerdings ist sicherzustellen, dass die cons-Zellen eines Argumentes nicht verändert werden. Haben wir

```
(set-cdr! (Quasi-letzte-Element
            Zirkuläre_Liste          ;Listenzeiger
            (cdr Zirkuläre_Liste)    ;aktueller Zeiger
            (cdr Zirkuläre_Liste))   ;vorhergehende Position
          (cdr Zirkuläre_Liste))
```




```
(set-car! Zirkuläre_Liste null          ;notwendig, falls zirkuläre Liste
                                        ;nur aus einer cons-Zelle besteht.
(set! Zirkuläre_Liste (cdr Zirkuläre_Liste))
```



Legende:
↪ Symbole S. 165; ↪ Programmcode S. 170

Abbildung 2.10: Vorgang aus der zirkulären Liste entfernen

keinen Einfluss auf ein Konstrukt, d. h. können wir seine Implementation nicht beeinflussen und wollen trotzdem sichergehen, dass die `cons`-Zellen des Arguments nicht modifiziert werden, dann können wir diese nicht direkt dem fremden Konstrukt übergeben. Keine `lambda`-Variable darf direkt an diese `cons`-Zellen gebunden werden. Es ist zunächst eine Kopie des Argumentes durch neue `cons`-Zellen zu konstruieren und diese Kopie dient dann als Bindungswert. Wird die Kopie verändert, bleibt das Original erhalten. Die Sicherstellung des Argumentes ist damit erreicht. Eine solche Kopie ist der Rückgabewert des `copy`-Konstruktes.

Fall 1: Modifikation der `cons`-Zellen des Argumentes

```
eval> (define Foo '(A (B C) D))
eval> (define Baz Foo) ;Foo ist Argument
                       ; für das
                       ; define-Konstrukt
eval> (eq? Foo Baz) ==> #t ;Foo und
                      ; Baz zeigen auf
                      ; dieselben cons-Zellen
eval> (set-cdr! Baz '(E))
eval> Baz ==> (a e)
eval> Foo ==> (a e)   ;Modifikation von Foo !
```

Fall 2: Modifikationsschutz durch `list-copy`-Konstrukt

Hinweis: `list-copy` in *PLT-Scheme* (*DrScheme*)
Das Konstrukt `list-copy` steht in *PLT-Scheme* (*DrScheme*) zur Verfügung, wenn die Defintionen von `list.ss` geladen und compiliert wurden. Bei der Standardinstallation von *PLT-Scheme* (*DrScheme*) steht `list.ss` unter:
`C:\Programme\plt\collects\srfi\1\list.ss`
anders formuliert: `eval> (require srfi/1/list)`
(Achtung! Passt aber nicht mit der Spracheinstellung `R5RS` zusammen — daher wenn auch `set-car!` und/oder `set-cdr!` mit:

```
eval> (require rnrs/base-6)
eval> (require rnrs/mutable-pairs-6)
```

```
;;;;Für das list-copy
eval> (require srfi/1/list)

eval> (define Foo '(A (B C) D))
eval> (define Baz (list-copy Foo))
eval> (eq? Foo Baz) ==> #f ;Foo und Baz
                           ; zeigen auf
                           ; verschiedene
                           ; cons-Zellen
eval> (equal? Foo Baz) ==> #t
```

```
eval> (set! Foo "Alles klar?")
eval> Foo ==> "Alles klar?"
eval> Baz ==> (A (B C) D)
eval> (set! Baz "OK!")
eval> Foo ==> "Alles klar?"
eval> Baz ==> "OK!"
```

Fall 3: Zirkuläre Liste

```
;;;;Notwendige Bibliotheken in PLT-Scheme
;;; für die zirkuläre Liste
eval> (require rnrs/base-6)
eval> (require rnrs/mutable-pairs-6)
;;;;Für das list-copy
eval> (require srfi/1/list)

eval> (define Foo '(a (b c) d))
eval> (set-cdr! (cddr Foo) Foo)
eval> Foo
      ==> #0={a {b c} d . #0#}
eval> (define Baz Foo)
eval> (set-car! (cadr Baz) 'e)
eval> Baz
      ==> #0={a {e c} d . #0#}
eval> (car (cddddr Baz))
      ==> {e c}
eval> (car (cddddr Foo))
      ==> {e c}
eval> (define Bar (list-copy Foo))
eval> Bar
      ==> #0={a {e c} d . #0#}
eval> (set-car! (cadr Baz) 'f)
eval> Baz
      ==> #0={a {f c} d . #0#}
eval> Foo
     ==> #0={a {f c} d . #0#}
eval> Bar
     ==> #0={a {f c} d . #0#}
```

Bei einer zirkulären Liste ist das hier eingesetzte `list-copy`-Konstrukt nicht nützlich. Hier wäre die Kopie selbst mit Hilfe des `cons`-Konstruktes zu programmieren.

Wir halten fest: Wenn ein Listenelement verändert wird, können dadurch gleichzeitig andere Listen davon betroffen sein. Zu empfehlen ist daher, eine Liste, die als Konstante dient, prinzipiell nicht zu verändern (↪ [36], p. 23). Man stelle sich vor, eine solche Konstante befindet sich in einem Lesespeicher (engl.: *read only memory*). Jeder Änderungsversuch führt zu einem Hardwarefehler. Das folgende Beispiel macht noch-

mals die Gefahr deutlich, den Nebeneffekt der Modifikation zu übersehen.

```
;;;;Sprache: R5RS
eval> (define Parlaments-Fraktion?
       (lambda (x)
        (member x
          '(SPD CDU FDP Gruene Linke))))
eval> (define so-nicht!
       (lambda (x)
        (let
         ((Teil (Parlaments-Fraktion? x)))
         (cond (Teil
                (set-car! Teil
                          (list)))
               (#t (list))))))

eval> (Parlaments-Fraktion? 'FDP)
      ==> (fdp gruene linke)
eval> (so-nicht! 'FDP)
eval> (Parlaments-Fraktion? 'FDP)
      ==> #f
```

2.2.2 Assoziationsliste

Wenn wir auf der Basis der einfachen Liste Konstruktoren, Selektoren, Prädikate und Mutatoren definieren, dann ist vorab die Listenstruktur festzulegen, d. h. die Bedeutung der einzelnen Listenelemente ist zu fixieren. Wenn wir annehmen, dass es sich bei einer Liste um ein Konto handelt, dann ist beispielsweise festzulegen, dass das 1. Element die Kontonummer, das 2. Element den Kontentyp, das 3. Element den Namen des Kontoinhabers etc. abbilden. Wir betrachten Kontonummer, Kontentyp, Name des Kontoinhabers etc. als Attribute A_i zum Objekt Konto. Den einzelnen Listenelementen sind dann entsprechend ihrer Reihenfolge die einzelnen Attributbeschreibungen zuzuordnen.

Objekt Konto: $(< a_1 >< a_2 >< a_3 >< a_4 > \ldots < a_n >)$

Beschreibung der Attribute:

Wert in der Liste		Bedeutung
$< a_1 >$	$\equiv$	Kontonummer
$< a_2 >$	$\equiv$	Kontentyp
$< a_3 >$	$\equiv$	Name des Kontoinhabers
	$\vdots$	
$< a_n >$	$\equiv$	Datum der letzten Buchung

Hinweis: Begriff *Datenlexikon*.
Zur Beschreibung eines Attributes gehört in der Regel auch die Defini-

tion des Wertebereichs, z. B.: Kontonummer, 4-stellig numerisch, positiv ganzzahlig, $(and(<= 0 < a_1 >)(<=< a_1 > 9999))$, und zusätzliche textliche Erläuterungen z. B. 1. Stelle Kennziffer der Abteilung gemäß Organisationsdiagramm. Diese Informationen weist das Datenlexikon aus.

Exkurs: *Objekt* und *Wert*.
Geht es um die destruktive Änderung eines symbolischen Ausdruckes, dann bezeichnen wir ihn als ein Objekt. Symbolische Ausdrücke betrachten wir allgemein als Werte z. B. als Rückgabewert einer Funktion. Mathematisch gesehen, existiert ein Wert (engl.: *value*) und ist nicht änderbar; an seine Stelle kann nur ein neuer Wert treten.

Das Verstehen der anwendungsspezifischen Bedeutung eines Listenelementes setzt voraus, dass wir seine Position innerhalb der Liste kennen. Kurz gesagt: Die Position ist Informationsträger. Wir nehmen für das Konto z. B. an, dass folgende Selektoren definiert sind:

```
eval> (define Kontonummer
        (lambda (Liste) (list-ref Liste 0)))
eval> (define Kontentyp
        (lambda (Liste) (list-ref Liste 1)))
eval> (define Name-des-Kontoinhabers
        (lambda (Liste) (list-ref Liste 2)))
        .
        .
        .
eval> (define Datum-letzte-Buchung
        (lambda (Liste) (list-ref Liste n)))
```

Damit der jeweilige Selektor das richtige Attribut ermittelt, ist sicherzustellen, dass die Position eines Attributes sich in der Liste nicht verschiebt. Die `lambda`-Variable `Liste` der obigen Selektoren ist an ein Konto zu binden, bei dem die definierte Struktur statisch ist. Löschen eines Attributwertes setzt voraus, dass die Listenelemente ihre Position nicht ändern. Daher ist der aktuelle Attributwert beim Löschen durch einen vereinbarten Ersatzwert (engl.: *default value*) bzw. Löschwert, z. B. `null` oder `(list)`, zu ersetzen. Die Maxime „Position ist Informationsträger“ ist statisch und daher nur bedingt geeignet für das prinzipiell dynamische Listenkonzept. Dynamisch ist die Liste in Bezug auf das Verschieben von Elementen, beim Einfügen und Löschen von Elementen.

Sind viele Attribute abzubilden, dann ist die einfache Liste kaum geeignet. Ihr dynamisches Konzept erweist sich als nachteilig, wenn von den einzelnen Objekten (hier: Konten) nur wenige Attribute einen Wert ungleich dem Ersatzwert haben. In einem solchen Fall hat die Liste viele Platzhalter, die nur deshalb erforderlich sind, um die fest vorgegebene Position der Elemente zu sichern. Dadurch wird erstens die Selektionszeit verlängert und zweitens das Verstehen des Programmes unnötig erschwert.

Zumindest bei wenig „besetzten“ Attributen ist es angebracht, die Position in einer Liste nicht als Informationsträger zu nutzen. Die Information, um welches Attribut es sich handelt, kann dem Element zugeordnet werden, wenn die Elemente jetzt selbst Listen sind. Eine solche Verknüpfung des Attributs mit seinem Wert wird als *Assoziationsliste*, kurz: *A-Liste*, bezeichnet. Eine A-Liste ist eine Liste mit eingebetteten Sublisten, in denen das erste Element jeweils die Aufgabe eines Schlüsselbegriffs (*Key*) übernimmt.

Hinweis: *A-Liste*.
Allgemein können A-Listen als Listen aufgefasst werden, deren Elemente Punkt-Paare (engl.: *dotted pairs*) sind.

In unserem Beispiel könnte das Konto dann folgende Listenstruktur haben:

Objekt Konto:

```
((Kontonummer             < a1 >)
 (Kontotyp                < a2 >)
 (Name-des-Kontoinhabers  < a3 >)
                          ⋮
 (Datum-letzte-Buchung    < an >))
```

Zur Konstruktion der Selektoren, die auf der Basis A-Liste arbeiten, können wir eingebaute Zugriffskonstrukte nutzen z. B. in Scheme `assoc`, `assq` oder `assv`. Solche eingebauten Konstrukte sorgen für eine effiziente Selektion, da sie die jeweiligen Implementationsdetails nutzen können. Ihre prinzipielle Funktion verdeutlicht das selbstdefinierte Konstrukt `Mein-assq`:

```
eval> (define Mein-assq
       (lambda (Key A_Liste)
        (cond ((null? A_Liste) null)
              ((eq? Key (caar A_Liste))
               (car A_Liste))
              (#t (Mein-assq Key
                   (cdr A_Liste))))))
```

In dem Konstrukt `Mein-assq` fußt die Vergleichsoperation zwischen dem Wert, an den `Key` gebunden ist, und dem ersten Element einer Subliste auf dem `eq?`-Konstrukt. Unterstellen wir, dass der Wert von `Key` nicht nur ein Symbol sein kann, sondern z. B. eine Liste, dann ist für die Vergleichsoperation das `equal?`-Konstrukt zu verwenden. In solchem Fall wollen wir die Subliste selektieren, deren erstes Element „gleich“ der Liste von `Key` ist. Da hier Gleichheit nicht Identität bedeutet (d. h. nicht die identischen `cons`-Zellen für beide Listen voraussetzen soll), ist `equal?` statt `eq?` zu wählen. Die Benutzung der eingebauten Selektoren für A-Listen setzt daher die Kenntnis der Vergleichsoperation voraus.

Passende Selektoren für das Kontobeispiel können mit dem `Mein--assq`-Konstrukt, wie folgt, definiert werden:

```
eval> (define Kontonummer
        (lambda (A_Liste)
          (Mein-assq 'Kontonummer A_Liste)))
eval> (define Kontotyp
        (lambda (A_Liste)
          (Mein-assq 'Kontotyp A_Liste)))
eval> (define Name-des-Kontoinhabers
        (lambda (A_Liste)
          (Mein-assq 'Name-des-Kontoinhabers
                     A_Liste)))
```

Hinweis: *Subliste* als Selektionswert.
Die A-Listen-Selektoren haben im Trefferfall die ganze Subliste als Wert und nicht nur den zum Schlüssel assoziierten Wert. Vermieden wird damit die Doppeldeutigkeit im Falle `null`. Ist ein Schlüssel mit dem Wert `null` verknüpft, dann wäre sonst der Rückgabewert bei einigen LISP-Systemen gleich dem Wert „kein Treffer“.

Eine A-Liste, die z. B. keine Eintragung für `Kontotyp` enthält, kann im Gegensatz zur einfachen Listenstruktur nun problemlos als Argument für den Selektor `Name-des-Kontoinhabers` dienen.

```
eval> (define *K007* '((Kontotyp 'Sachmittel)
                       (Kontonummer 4711)))
eval> (Name-des-Kontoinhabers *K007*)
      ==> ()
eval> (Kontonummer *K007*)
      ==> (Kontonummer 4711)
```

Zum Konstruieren und Ändern einer A-Liste stehen dieselben Konstrukte wie für die einfache Liste zur Verfügung: `cons`, `list`, `set-car!`, `set-cdr!` etc. Auf die A-Liste ausgerichtete Konstruktoren und Mutatoren sind selbst zu definieren. Ein Prädikat zum Erkennen einer A-Liste kann, wie folgt, definiert werden:

```
eval> (define A-Liste?
        (lambda (Objekt)
          (letrec
              ((Leere-A-Liste?
                (lambda (x)
                  (equal? x (list null))))
               (Nur-eine-Subliste?
                (lambda (x)
                  (and
                   (pair? x)
                   (null? (cdr x))
                   (pair? (car x)))))
               (A-L?
```

```
                (lambda (x)
                  (cond
                    ((or (not (pair? x))
                         (null? x))
                     #f)
                    ((or (Leere-A-Liste? x)
                         (Nur-eine-Subliste? x))
                     #t)
                    ((pair? (car x))
                     (A-L? (cdr x)))
                    (#t #f)))))
            (A-L? Objekt))))
eval> (A-Liste? '((a 10) (b 9) (c 3)))
      ==> #t
eval> (A-Liste? '(()))
      ==> #t
eval> (A-Liste? '(a (b) (c 3)))
      ==> #f
eval> (A-Liste? (lambda (X) (+ 1 2)))
      ==> #f
```

Bei der Definition eines A-Listen-Konstruktors ist zu entscheiden, ob die A-Liste dasselbe Attribut (denselben Schlüsselbegriff) mehrfach enthalten kann oder nicht (*Multiplizität*). Der oben skizzierte Selektor `Mein-assq` wäre nicht in der Lage, eine zweite Ausprägung desselben Attributs zu selektieren, wie das folgende Beispiel zeigt.

```
eval> (define *K001*
        '((Strasse "An der Eulenburg 6")
          (Ort "Reppenstedt")
          (PLZ 21391)
          (Telefon "04131/63845")))
eval> (define Cons-A-Liste
        (lambda (Neue_Element A_Liste)
          (cond ((pair? Neue_Element)
                 (cons Neue_Element A_Liste))
                (#t (cons
                     (list Neue_Element)
                     A_Liste)))))
eval> (set! *K001*
            (Cons-A-Liste
             '(Telefon "01626989027")
             *K001*))
eval> *K001* ==>
      ((Telefon "01626989027")
       (Strasse "An der Eulenburg 6")
       (Ort "Reppenstedt")
       (PLZ 21391)
       (Telefon "04131/63845"))
```

```
eval> (Mein-assq 'Telefon *K001*)
      ==> (Telefon "01626989027")

eval> (assq 'Telefon *K001*)
     ==> (Telefon "01626989027")
```

Die Möglichkeit, A-Listen mit Attributen zu definieren, die mehrfach vorkommen können, setzt eine problemgerechte Festlegung für den Verbund von Konstruktoren, Selektoren, Prädikaten und Mutatoren voraus.

Beispielsweise eignen sich A-Listen mit Multiplizität zum Abbilden von aktuellen Werten gemeinsam mit ihren vorherigen Werten (d. h. ihrer Historie). Im Fall der globalen Variable `*K001*` bleibt die alte Telefonnummer bestehen. Existierende Konstrukte greifen mit ihrem Selektor `Telefon` ohne Änderung auf die aktuelle Telefonnummer zu. Die Historie, d. h. die zunächst unter dem Attribut `Telefon` gespeicherte Telefonnummer, kann z. B. für Revisionszwecke über einen entsprechend definierten Selektor wieder gewonnen werden.

Sind die Schlüssel einer A-Liste Symbole und kommt jeder Schlüssel nur einmal in der A-Liste vor, dann bietet sich als Alternative die Eigenschaftsliste an (↪ Abschnitt 2.2.3 S. 191). Sind die Schlüssel nicht nur Symbole, dann ist in modernen LISP-Systemen die *Hash*-Tabelle eine Alternative. Für große A-Listen bieten sie eine Effizienzsteigerung, da die Zugriffszeit auf eine Assoziation (quasi) konstant ist. Wir skizzieren die Hash-Option für *PLT-Scheme* im folgenden Exkurs.

Exkurs: *Hash*-Tabelle in *PLT-Scheme*

`hashtable` ist eine Datenstruktur, die Schlüssel (*Keys*) mit Werten (*Values*) verknüpft. Zur Nutzung in *PLT-Scheme* ist die besondere Bibliothek `hashtables-6` erforderlich, d. h.:

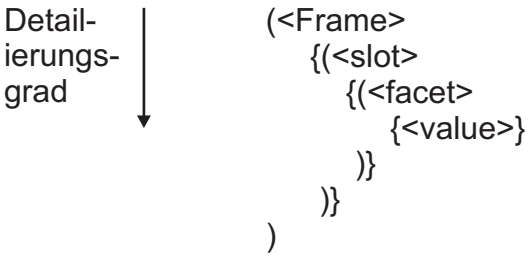
```
eval> (require rnrs/hashtables-6)
```

In dieser *Hash*-Tabelle kann jedes Objekt als *Key* dienen, vorausgesetzt es gibt dafür:

1. Eine Funktion, die das Objekt in ein exaktes Integerobjekt überführt (≡ *hash function*).
2. Eine Funktion, die einen nachfragenden *Key* mit dem in der Tabelle auf Gleichheit überprüft (≡ *equivalence function*).

Für Standardfälle gibt es einen eingebauten Konstruktor `make-eq-hashtable`, der mit `eq?` auf Gleichheit testet. Das folgende Beispiel zeigt einige Konstrukte:

```
eval> (define ht (make-eq-hashtable))
eval> (hashtable-set! ht
       'A "lambda Das ultimative Denkmodell")
eval> (hashtable-set! ht
       "More" '(Means Of Requirements Engineering))
eval> (hashtable-set! ht
       #\a '(lambda (x) (print x) x))
eval> (hashtable-set! ht
       4711 "Manchmal sehr notwendig")
eval> (hashtable-set! ht
       '(A B C) '(1 2 3 4 5 6 7 8 9 1 2 3 4))
eval> (hashtable-set! ht
       #(Bonn Berlin) "Hauptstadt von?")
eval> (hashtable-entries ht) ==>
      ;Zwei Werte: Keys und Values

      #("More" #\a 4711 A (A B C) #(Bonn Berlin))

      #((Means Of Requirements Engineering)
      (lambda (x) (print x) x)
      "Manchmal sehr notwendig"
      "lambda Das ultimative Denkmodell"
      (1 2 3 4 5 6 7 8 9 1 2 3 4)
      "Hauptstadt von?")

eval> (hashtable? ht) ==> #t
eval> (hashtable-size ht) ==> 6
eval> (hashtable-ref ht
       #\a "Wenn nicht dabei!")
      ==>
     (lambda (x) (print x) x)
eval> (hashtable-ref ht
```



Legende:
BNF-Notation (↪ Abschnitt 1.2.2 S. 66)
< *frame* > ≡ Rahmen-Name; oberste Ebene der Einheit
< *slot* > ≡ Einschub-Name; nächste Ebene (Komponenten)
< *facet* > ≡ Facetten-Name; Attribute einer Komponente
< *value* > ≡ Wert; Ausprägungen der Attribute

Abbildung 2.11: Rahmen konstruiert aus geschachtelten A-Listen

```
        #\b "Wenn nicht dabei!")
       ==>
       "Wenn nicht dabei!"
eval> (hashtable-contains? ht 'A)
       ==> #t
eval> (hashtable-contains? ht '(A B C))
       ==> #f ;Da: (eq? '(A B C) '(A B C))
                  ==> #f
eval> (string-hash "More")
       ==> 75413158
eval> (symbol-hash 'A)
       ==> 80027
eval> (equal-hash '(A B C))
       ==> 629107430
```

Aus geschachtelten A-Listen sind Rahmen (engl.: *frames*) konstruierbar, die eine mehrstufige Struktur darstellen (↪ Abbildung 2.11 S. 190). Mit solchen Rahmen hat *Marvin Minsky* ein Paradigma zur Abbildung von „Wissen“ vorgeschlagen (↪ [128]). Ein Rahmen besteht aus Einschüben (engl.: *slots*). Die Einschübe haben Facetten (engl.: *facets*), die wiederum Werte haben, die voreingestellt sein können. Die Werte selbst können weiter strukturiert sein, aber auch Funktionen umfassen.

Wir können z. B. ein Konto als einen Rahmen definieren. Alle Rahmen fassen wir hier in einer A-Liste `*RAHMEN-DATENBANK*` zusammen. Es sei angemerkt, dass es zweckmäßiger ist, den einzelnen Rahmen als Eigenschaft einer Kontonummer zu verwalten. Die Konstrukte für eine solche Bindung behandeln wir im Abschnitt 2.2.3 S. 191.

```
eval> (define *RAHMEN-DATENBANK*
        '(Rahmen
```

```
(K001
 (Kontotyp (Gruppe Sachkonto))
 (Kontoinhaber (Name Krause)
               (Telefon 4711))
 (Letzte-Buchung
  (Datum 20101127)
  (Buchungsart Auszahlung)
  (EUR 5.0))
 (Einzahlung
  (EUR-Datum (10.0 20101127)
             (55.0 20101126)))
 (Auszahlung
  (EUR-Datum (5.0 20101124)))
 (Saldo (EUR 60.0)))
(K002
 (Kontotyp (Gruppe Haushalt))
 (Kontoinhaber (Name Eiser))
 (Letzte-Buchung
  (Datum 20101127)
  (Buchungsart Einzahlung)
  (EUR 105.0))
 (Einzahlung
  (EUR-Datum (105.0 20101127)))
 (Saldo (EUR 105.0)))
))
```

2.2.3 Eigenschaftsliste

Die Eigenschaftsliste, kurz: P-Liste (*Property List*) genannt, ist das klassische Konzept zur Abbildung eines Lexikons. Mit der Definition eines Symbols (Literalatoms) wird in den klassischen LISP-Systemen eine an das Symbol gebundene Eigenschaftsliste verfügbar. Jedes Symbol hat eine P-Liste, die zunächst leer ist. Ziehen wir eine Analogie zur Kernspaltung in der Physik, dann entsprechen den Neutronen, Elektronen, Protonen, die Zellen des (Literal-)Atoms. Abhängig vom jeweiligen LISP-System und seiner Implementation fasst das Symbol bestimmte Zellen (*slots*) zusammen, d. h. einem Symbol sind Speichermöglichkeiten für mehrere Zwecke zugeordnet (↪ Abbildung 2.20 S. 249).

Hinweis: `putprop`, `getprop` in *PLT-Scheme* (*DrScheme*)
Die Konstruke zur P-Liste sind in `MzScheme` vorhanden. `MzScheme` steht in *PLT-Scheme* (*DrScheme*) zur Verfügung, wenn die Defintionen von `compat.ss` geladen und compiliert wurden. Bei der Standardinstallation von *PLT-Scheme* (*DrScheme*) steht der `module compat` unter:
`C:\Programme\plt\collects\mzlib\compat.ss`
anders formuliert:

```
eval> (require mzlib/compat)
```

Bisher haben wir den Wert eines Symbols betrachtet. Dieser wird mit der „Wertzelle“ (engl.: *value cell*) abgebildet. Beim Evaluieren eines Symbols (↪ EVAL-Regel 4, S. 26) wird auf diese Zelle zugegriffen. Im allgemeinen verfügt ein Symbol zusätzlich über eine Zelle zum Abbilden der P-Liste.

Eine Eintragung in die P-Liste eines Symbols erfolgt mittels `putprop`. Der zugehörige Selektor ist das `getprop`-Konstrukt. Mit dem `remprop`-Konstrukt wird eine Eigenschaft von der P-Liste entfernt. Die gesamte Eigenschaftsliste ist durch das `proplist`-Konstrukt verfügbar. Ein besonderes Prädikat für die P-Liste gibt es nicht, da sich die P-Liste von einer „normalen“ Liste nicht unterscheidet. Zu beachten ist, dass manche Implementationen keinen Rückgabewert ausgeben; d. h. es gibt kein `==>` $<$ *wert* $>$.

Konstruktor/Mutator: (Eingeschränkte Verfügbarkeit in *PLT-Scheme*[7])

`eval>` (`putprop` $<$ *symbol* $><$ *eigenschaft* $><$ *wert* $>$)
`==>` $<$ *wert* $>$[8]

`eval>` (`remprop` $<$ *symbol* $><$ *eigenschaft* $>$)
`==>` $<$ *symbol* $-$ *p* $-$ *liste* $>$

Selektoren: (Eingeschränkte Verfügbarkeit in *PLT-Scheme*[9])

`eval>` (`proplist` $<$ *symbol* $>$) `==>` $<$ *p* $-$ *liste* $>$;ganze P-Liste
`eval>` (`getprop` $<$ *symbol* $><$ *eigenschaft* $>$) `==>` $<$ *wert* $>$

Prädikat:

`eval>` (`pair?` $<$ *p* $-$ *liste* $>$) `==>` #t

mit:

[7] Das `remprop`-Konstrukt ist nicht im *Module* `compat mzscheme` enthalten. Ein Ersatz lässt sich aber wie folgt definieren: `(define remprop (lambda (Symbol Key)(putprop Symbol Key #f)))`

[8] Die Reihenfolge von $<$ *wert* $>$ und $<$ *eigenschaft* $>$ ist oft unterschiedlich implementiert.

[9] Das `proplist`-Konstrukt ist nicht im *Module* `compat mzscheme` enthalten.

$<$ *symbol* $>$	$\equiv$ Das Symbol, dessen Eigenschaftsliste betroffen ist.
$<$ *wert* $>$	$\equiv$ Beliebiger symbolischer Ausdruck als Ausprägung der Eigenschaft.
$<$ *eigenschaft* $>$	$\equiv$ Ein Symbol, das die Eigenschaft von $<$ *symbol* $>$ benennt.
$<$ *p − liste* $>$	$\equiv$ Eine Liste mit der Elementfolge aus $<$ *eigenschaft* $>$ und $<$ *wert* $>$, sortiert in der umgekehrten Reihenfolge der `putprop`-Applikationen.
$<$ *symbol − p − liste* $>$	$\equiv$ Modifizierte $<$ *p − liste* $>$ mit $<$ *symbol* $>$ als zusätzliches erstes Element.

Die Konstrukte `putprop`, `getprop`, `remprop` und `proplist` evaluieren ihre Argumente, so dass an der Stelle von $<$ *symbol* $>$, $<$ *eigenschaft* $>$ und $<$ *wert* $>$ symbolische Ausdrücke stehen, die einen entsprechenden Rückgabewert haben.

Das folgende Beispiel verdeutlicht die Konstrukte zur Eigenschaftsliste[10]:

```
eval> (define *K001* "Allgemeine Daten")
eval> (pair? *K001*) ==> #f
eval> (putprop '*K001* 'Ort "Reppenstedt")
eval> (putprop '*K001* 'PLZ 21391)
eval> (getprop '*K001* 'Ort)
      ==> "Reppenstedt"
eval> (getprop '*K001* 'Strasse) ==> #f
eval> (putprop '*K001* 'Zufahrt #f)
eval> (getprop '*K001* 'Zufahrt)
      ==> #f ;Achtung!
             ; Keine Unterscheidung zwischen
             , einer undefinierten Eigenschaft
             ; und einer definierten Eigenschaft
             ; mit dem Wert #f.
```

In klassischen LISP-Systemen (↪ z. B. [124]) wird die P-Liste mitverwendet, um eine Funktion, die das Symbol benennt, zu speichern. Die Selektion des Funktionswertes eines Symbols entspricht einer Anwendung des `putprop`-Konstruktes mit der Eigenschaft `value` (↪ Abschnitt 2.4.3 S. 248).

In modernen LISP-Systemen ist die Wertzelle von der Zelle für die P-Liste zu unterscheiden. So kann ein Symbol in einer Umgebung keinen Wert haben, also keinen Eintrag in seiner Wertzelle aufweisen, jedoch Eintragungen in seiner P-Liste haben. Das folgende Beispiel zeigt einen solchen Fall.

```
eval> (define Modul-P
        (lambda()
```

[10] In *PLT-Scheme* mit *Module* `compat mzscheme`.

```
          (let((Schulze 'P1234)
               (Krause 'P5678))
            (putprop Schulze 'Beruf 'Manager)
            (putprop Krause 'Beruf 'Schreibkraft))
      #t))
eval> (Modul-P) ==> #t
eval> (getprop 'P1234 'Beruf)
      ==> Manager
eval> (getprop 'P5678 'Beruf)
      ==> Schreibkraft
eval> P1234 ==> ERROR ...
                ;reference to an identifier
                ; before its definition: P1234
```

Hinweis: Eigenschaft *Primop-Handler*.
Manche LISP-Systeme stellen einen sogenannte Primop-Handler zur Verfügung, z. B. *PC Scheme*. Es wird dann festgestellt, ob ein Symbol eine vorgegebene Eigenschaft, hier `PCS*PRIMOP-HANDLER`, hat; wenn ja, dann wird dessen `cdr`-Teil an die Stelle des Namens gesetzt. Das folgende Beispiel verdeutlicht die Auswertung der Eigenschaft `PCS*PRIMOP-HANDLER`.

```
PC-eval> (define FOO (lambda (X) (+ X 7))) ==> FOO
PC-eval> (FOO 4) ==> 11
PC-eval> (putprop 'FOO
          '("Hier ohne Bedeutung"
            lambda (Y) (+ Y 3)
           'PCS*PRIMOP-HANDLER)
        ==> ("Hier ohne Bedeutung" lambda (Y) (+ Y 3))
PC-eval> (FOO 4) ==> 7
PC-eval> FOO ==> (lambda (Y) (+ Y 3))
```

Mit Hilfe des Iterationskonstruktes `for-each` (↪ S. 89) ist von einer vorgegebenen A-Liste eine P-Liste einfach erstellbar. Dazu definieren wir das Konstrukt `Make-Lexikon`:

```
eval> (define Make-Lexikon
        (lambda (Symbol A_Liste)
          (for-each
           (lambda (x)
             (putprop Symbol
                      (car x)
                      (list-ref x 1)))
           A_Liste)))
eval> (define My-A-Liste
        '((Telefon "04131/63845")
          (Ort "Reppenstedt")
          (Telefon "01626989027")))
eval> (Make-Lexikon '*K002* My-A-Liste)
eval> (getprop '*K002* 'Telefon)
      ==> "01626989027"
      ;Zu beachten ist die Reihenfolge
      ; der Elemente. Sie weist den
```

```
      ; letzten Wert für die Eigenschaft
      ; Telefon aus, jedoch aufgrund
      ; des ersten putprop für
      ; Telefon als letztes
      ; P-Listen-Element. Neue
      ; Eigenschaften werden vorn eingefügt.
```

Die Konstruktion mit P-Listen birgt die Gefahr ungewollter Nebeneffekte. Das folgende Beispiel zeigt die Unterschiede der Effekte (Reichweite) der P-Liste zu einer lokalen A-Listendefinition. In beiden Fällen konstruieren wir ein Lexikon `Foo`, das zunächst die Attribute `Key-1`, `Key-2` und `Key-3` enthält. Der Wert von `Key-1` wird modifiziert. Danach wird auf der *top level*-Ebene mit dem gleichen lokalen Namen eine `Key-4`-Definition hinzugefügt.

Beispiel: Assoziationsliste (In *PLT-Scheme* mit Sprache `R5RS`)

```
eval> (define Lokale-A-Liste
        (lambda()
          (let ((A-Liste '((Key-1 value-1)
                           (Key-2 value-2)
                           (Key-3 value-3))))
            (set-cdr! (assq 'Key-1 A-Liste)
                      (list 'value-neu))
                A-Liste)))
eval> (define Foo (Lokale-A-Liste))
eval> Foo ==> ((key-1 value-neu)
               (key-2 value-2)
               (key-3 value-3))

eval> (define A-Liste '((Key-4 value-4)))

eval> (define Foo (Lokale-A-Liste))

eval> Foo ==> ((key-1 value-neu)
               (key-2 value-2)
               (key-3 value-3))
```

Beispiel: Eigenschaftsliste (In *PLT-Scheme* mit Sprache `Module` und `module compat mzscheme`)

```
eval> (define Vermeintlich-lokale-P-Liste
        (lambda ()
          (let ((Meine-P-Liste 'Baz))
            (putprop Meine-P-Liste
                     'Key-1 'value-1)
            (putprop Meine-P-Liste
                     'Key-2 'value-2)
```

```
            (putprop Meine-P-Liste
                     'Key-3 'value-3)
            (putprop Meine-P-Liste
                     'Key-1 'value-neu)
            Meine-P-Liste)))
eval> (define Foo (Vermeintlich-lokale-P-Liste))
eval> Foo ==> Baz
eval> (getprop Foo 'Key-1) ==> value-neu
eval> (getprop Foo 'Key-4) ==> #f
eval> (putprop 'Baz 'Key-4 'value-4)
eval> (getprop Foo 'Key-4)
      ==> value-4 ;Achtung Nebeneffekt!
                  : Key-4 existiert jetzt.
```

Die P-Liste ist für das Symbol `Baz` angelegt. Mit dem Definieren des `Foo`-Konstruktes zeigt das Symbol `Foo` auf die P-Liste von `Baz`. Wird die P-Liste von `Baz` verändert, dann ist `Foo` davon mitbetroffen. Diese Globalität der P-Liste führt leicht zu ungewollten Nebeneffekten (Fehlerquelle!). Man kann ihre Globalität jedoch vorteilhaft zur Kommunikation zwischen Konstrukten nutzen.

Eine Kommunikation über die P-Liste zeigt das folgende Beispiel `Schnittmenge-Symbol`. Wir berechnen die Schnittmenge von Symbolen in zwei Listen. Zunächst werden alle Symbole, die in der ersten Liste vorkommen, markiert. Sie erhalten eine Eintragung in ihrer P-Liste. Damit eine bereits vorhandene Eigenschaft nicht überschrieben wird, ist sicherzustellen, dass wir eine Eigenschaft wählen, die nicht noch einmal vorkommt. Dazu nutzen wir das `gensym`-Konstrukt (Näheres dazu ↪ Abschnitt 2.4.4 S. 259). Die zweite Liste wird Symbol für Symbol abgearbeitet. Für jedes Symbol prüfen wir, ob es markiert ist. Wenn es markiert ist, dann gehört es zur Schnittmenge. Da es in der zweiten Liste nochmals vorkommen kann, heben wir im Trefferfall die Markierung wieder auf. Alle nicht markierten Symbole gehören nicht zur Schnittmenge. Zum Schluss heben wir die noch bestehenden Markierungen von Symbolen der ersten Liste auf.

```
eval> (define Konten
        '(a b c d e f g
          (h i j k) l m n o p))
eval> (define Offene-Posten
        '(f z y g g a c))
eval> (define remprop ; Ersatzkonstrukt
        (lambda (Symbol Key)
          (putprop Symbol Key #f)))
eval (define Schnittmenge-Symbol
      (lambda (Liste_1 Liste_2)
        (letrec ((Marke (gensym))
          (Markieren
            (for-each
              (lambda (x)
```

```
                (cond
                 ((symbol? x)
                  (putprop x Marke #t))
                 (#t #t)))
               Liste_1))
              (Raus-Unmarkiert
               (lambda (Liste)
                 (cond ((null? Liste) (list))
                       ((and
                         (symbol? (car Liste))
                         (getprop
                          (car Liste) Marke))
                        ;Im markierten Fall
                        ; Markierung zurueck-
                        ; setzen. Dadurch keine
                        ; doppelten Symbole in
                        ; der Ausgangsliste.
                        (remprop (car Liste) Marke)
                         (cons (car Liste)
                               (Raus-Unmarkiert
                                (cdr Liste))))
                       (#t (Raus-Unmarkiert
                                (cdr Liste)))))))
          (begin0
            (Raus-Unmarkiert Liste_2)
             ;;Löschen noch bestehender Marken
             ;; in der Liste_1
            (for-each
             (lambda (x)
               (cond ((and (symbol? x)
                           (getprop x Marke))
                      (remprop x Marke))
                     (#t #t)))
              Liste_1)))))
eval> (Schnittmenge-Symbol Konten Offene-Posten)
      ==> (f g a c)
```

Als weiteres Beispiel für die Abbildungsoption P-Liste betrachten wir im folgenden ein Bürogebäude (↪ Abbildung 2.12 S. 200). Die Akten (Geschäftsvorfälle) werden in Laufmappen transportiert. Den Transport übernimmt ein mechanisches Fließbandsystem. Zu seiner elektronischen Steuerung ist ein Weg zwischen der jeweiligen Startstation und der angegebenen Zielstation zu bestimmen. Aufgabe des LISP-Programmes ist es, einen solchen Weg zu ermitteln.

Beispiel: Laufmappen-Transportsteuerung

Entsprechend der Büroskizze (↪ Abbildung 2.12 S. 200) ist für jede Hauspoststation eine P-Liste definiert (↪ Programm in Tabelle 2.5 S. 199). Sie enthält:

1. als Eigenschaft `Ort`, die Lage der Poststation, angegeben in *x*-*y*-Koordinaten, und
2. als Eigenschaft `Nachfolger` die Hauspoststationen, die direkt erreicht werden können sowie ihre Entfernung (als A-Liste).

Wir unterstellen, dass stets nur eine Laufmappe transportiert wird und die einzelnen Stationen diese Laufmappe durchschleusen bzw. aufnehmen können. Wir reduzieren damit das Problem auf eine einfache Wegesuche. Zunächst sei darüber hinaus nur gefordert, dass ein möglicher Weg zwischen Start und Ziel gefunden wird. Dieser Weg muss nicht der kürzeste Weg sein.

Als Problemlösung bilden wir das Verfahren *Tiefensuche* (engl.: *depth-first-search*) ab. Die Bezeichnung „Tiefe“ geht von einem Suchbaum (azyklischen Grafen) aus, der auf dem Kopf steht. Sie verdeutlicht, dass dieser Baum zunächst in Richtung auf die „Blätter“ untersucht wird, ehe der nächste „Zweig“ in Betracht gezogen wird. Gilt es z. B., einen Weg vom Startknoten `H` zum Zielknoten `F` zu finden, dann stellt Abbildung 2.13 S. 201 die Knoten dar, die bei der Tiefensuche abzuarbeiten („zu expandieren“) sind, ehe der Zielknoten `F` erreicht wird. Zunächst ist zu entscheiden, welcher Nachfolger vom Startknoten `H` als erster zu untersuchen ist. Ist `C` oder `G` zu wählen? Wir nehmen `C`, da `C` als erster Knoten in der Liste der Nachfolger von `H` genannt ist.

```
eval> (getprop 'H 'Nachfolger) ==> ((C 14) (G 17.5))
```

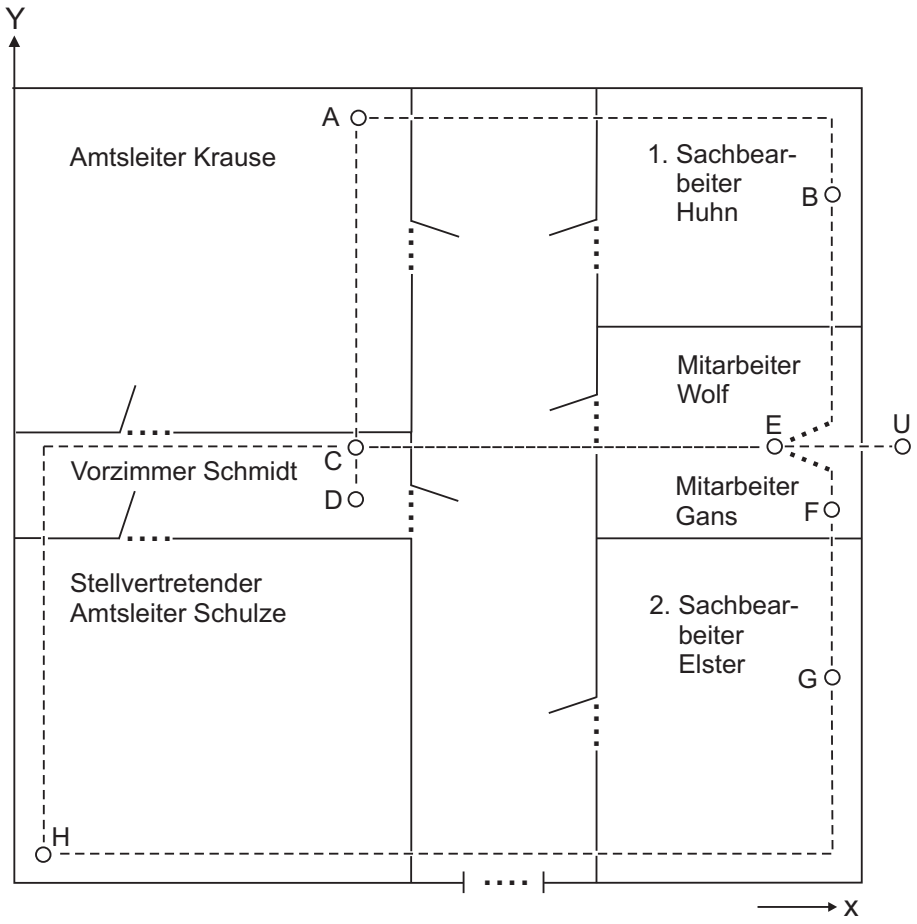
Da `C` nicht der Zielknoten ist, ist zu entscheiden, ob zunächst der erste Nachfolger von `C`, hier `D`, oder der bisher nicht geprüfte Knoten `G` zu analysieren ist? Im Sinne der „Tiefengewinnung“ verfolgen wir erst `D` weiter, also den ersten Nachfolger-Knoten („Kind“-Knoten) von `C`. Wir steigen im Suchbaum zunächst auf dem „linken Ast“ eine Ebene tiefer, vorausgesetzt, es gibt dort noch eine solche Ebene, bevor wir die anderen Knoten auf gleicher Ebene untersuchen. Knoten, die schon darauf überprüft wurden, dass sie nicht der Zielknoten sind, müssen dabei nicht noch einmal analysiert werden, sind also nicht mehr zu berücksichtigen.

Im Programm `Depth-first-Search` (↪ Programmcode S. 203) merken wir uns die noch zu expandierenden Knoten als eine einfache Liste mit dem Namen `Liste-eroeffnete-Knoten`. Knoten, die schon überprüft wurden, ob sie der Zielknoten sind, speichern wir als einfache Liste mit dem Namen `Liste-geschlossene-Knoten`. Um von den Nachfolgern eines expandierten Knotens diejenigen Knoten streichen zu können, die schon bearbeitet wurden, definieren wir eine Liste mit dem

```
eval> (define Buero (begin
      ;;;;Koordinaten der Knoten
          (putprop 'A 'Ort '(8.5 11.5))
          (putprop 'B 'Ort '(15.5 10))
          (putprop 'C 'Ort '(8.5 6.5))
          (putprop 'D 'Ort '(8.5 5.5))
          (putprop 'E 'Ort '(14 6.5))
          (putprop 'F 'Ort '(15.5 4.5))
          (putprop 'G 'Ort '(15.5 3))
          (putprop 'H 'Ort '(0.5 0.5))
          (putprop 'U 'Ort '(17 6.5))
     ;;;;Erreichbare Knoten und
     ;;;; ihre Entfernungen
          (putprop 'A 'Nachfolger
            '((C 5) (B 8.5)))
          (putprop 'B 'Nachfolger
            '((A 8.5) (E 4.5)))
          (putprop 'C 'Nachfolger
            '((D 1) (A 5) (E 5.5) (H 14)))
          (putprop 'D 'Nachfolger '((C 1)))
          (putprop 'E 'Nachfolger
            '((C 5.5) (F 3) (B 4.5) (U 3)))
          (putprop 'F 'Nachfolger
            '((E 3) (G 1.5)))
          (putprop 'G 'Nachfolger
            '((F 1.5) (H 17.5)))
          (putprop 'H 'Nachfolger
            '((C 14) (G 17.5)))
          (putprop 'U 'Nachfolger '((E 3)))
          "Ende"))
```

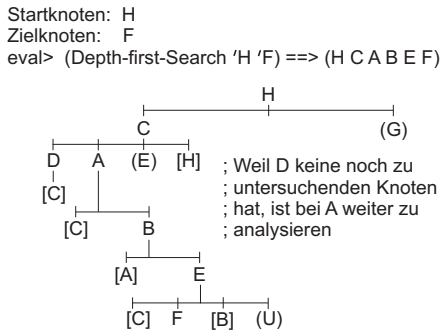
Legende:
↪ Abbildung 2.12 S. 200.

Tabelle 2.5: Programm: Hauspoststationen (Knoten) als P-Listen abgebildet



Legende:
-o- ≡ Hauspoststation x für den Laufmappentransport
--- ≡ Transportweg

Abbildung 2.12: Skizze der Büroräume und der Hauspoststationen



Legende:
(x) ≡ Noch zu analysierender Knoten
[x] ≡ Schon analysierter Knoten

Abbildung 2.13: Tiefensuche für Start H und Ziel F

Namen Arbeitsliste. Den aktuell expandierten Knoten speichern wir als Wert des Symbols Analyse-Knoten. Die Tabelle 2.6 S. 202 zeigt die Veränderung dieser Variablen bezogen auf das Durchlaufen der lokalen Funktion Tiefensuche. Darüber hinaus sind die putprop- und get-prop-Anwendungen angegeben.

Die Arbeitsliste bereinigen wir um die Knoten, die schon in der Liste-geschlossene-Knoten vorkommen. Dazu verwenden wir die Funktion Differenz-Knoten (↪ Tabelle 2.6 S. 202, Spalte Arbeits-liste, jeweils zweite Eintragung). Enthält die bereinigte Arbeits-liste Knoten, die auch in der Liste-eroeffnete-Knoten vorkommen, so streichen wir zunächst diese Knoten aus der Liste-eroeff-nete-Knoten. Dann setzen wir mit append die Knoten der bereinigten Arbeitsliste an den Anfang von Liste-eroeffnete-Kno-ten. Die zunächst aus Liste-eroeffnete-Knoten gestrichenen Kno-ten kommen dadurch wieder in die Liste-eroeffnete-Knoten, je-doch in der Reihenfolge von Arbeitsliste.

Zum Verstehen der Aufgabe sind vorab Anforderungen, Annahmen zum Entwurf und Testfälle formuliert. Dabei verweist die Abkürzung A auf eine *Anforderung*, die Abkürzung E auf eine Annahme zum Entwurf und die Abkürzung T auf einen *Testfall*.

A1: Depth-first-Search durchsucht ein angegebenes Knotennetz nach einem Weg von einem Start zu einem Ziel-Knoten.

- A1.1: Der erste selektierte Nachfolger wird zunächst weiter analysiert (expandiert), bevor die weiteren Nachfolger expandiert werden (Zweck: „Tiefengewinnung“).
- A1.2: Es sind ein Start- und ein Zielknoten anzugeben.

```
      eval> (Depth-first-Search 'H 'F) ==> (H C A B E F)
     Ana-          Liste-              Liste-ge-       Arbeits-
     ly-         eroeffnete-          schlossene-       lis-
     sek.          Knoten               Konten           te
      ()             (H)                  ()             ()
            (putprop 'H 'Weg '())
1.    H              ()                   (H)           (C G)
                                                        (C G)
                    (C G)
              (putprop 'C 'Weg 'H)
              (putprop 'G 'Weg 'H)
2.    C             (G)                  (C H)         (D A E H)
                                                        (D A E)
                  (D A E G)
              (putprop 'D 'Weg 'C)
              (putprop 'A 'Weg 'C)
              (putprop 'E 'Weg 'C)
.     D           (A E G)                (D C H)          (C)
                                                          ()
                  (A E G)
4.    A            (E G)                (A D C H)        (C B)
                                                          (B)
                  (B E G)
              (putprop 'B 'Weg 'A)
5.    B            (E G)                (B A D C H)      (A E)
                                                          (E)
                   (E G)
              (putprop 'E 'Weg 'B)
6.    E             (G)                (E B A D C H)    (C F B U)
                                                          (F U)
                  (F U G)
              (putprop 'F 'Weg 'E)
              (putprop 'U 'Weg 'E)
7.    F            (U G)              (F E B A D C H)     ; Ziel

    ... (append (Ausgabe-Weg (getprop 'F 'Weg)) (list 'F)) ...


                        ==> (H C A B E F)
```

Legende:
Reihenfolge der `putprop`- und `getprop`-Anwendungen im Programm `Depth-first-Search` bei Startknoten `H`, Zielknoten `F` und dem Büro von Programm in Tabelle 2.5 S. 199.

Tabelle 2.6: Reihenfolge im Programm `Depth-first-Search`

E1: Ein Knoten ist ein Symbol.

E1.1: Die Koordinaten eines Knotens sind als Eigenschaft Ort gespeichert.

E1.2: Die Nachfolger eines Knotens und ihre Entfernungen sind als Eigenschaft Nachfolger gespeichert und zwar als folgende A-Liste: ({(<nachfolger> <entfernung>)})

T1: Beispiel

T1.1: Daten (↪ Programm in Tabelle 2.5 S. 199)

T1.2: Applikation (↪ Tabelle 2.6 S. 202)

```
eval> (Depth-first-Search 'H 'F)
  ==> (H C A B E F)
```

Programm: Depth-first-Search

```
eval> (define Depth-first-Search
 (lambda (Start_Knoten Ziel_Knoten)
  (letrec
     ((Liste-eroeffnete-Knoten (list Start_Knoten))
      (Liste-geschlossene-Knoten (list))
      (Analyse-Knoten (list))
      (Arbeitsliste(list))
      (Get-Nachfolger ;vgl. E1.2 ohne
                      ;  Entfernungsangaben
       (lambda (Knoten)
         (map (lambda (X)
                (car X))
              (getprop Knoten 'Nachfolger))))
      ;;Gibt Liste_1 zurück ohne die Elemente,
      ;; die in Liste_2 vorkommen.
      (Differenz-Knoten
       (lambda (Liste_1 Liste_2)
         (cond((null? Liste_1) (list))
              ((member (car Liste_1) Liste_2)
               (Differenz-Knoten (cdr Liste_1)
                                 Liste_2))
              (#t (cons (car Liste_1)
                        (Differenz-Knoten
                                    (cdr Liste_1)
                                   Liste_2))))))
      ;;Die Eigenschaft Weg verknüpft
      ;; den Analyse-Knoten mit seinen Nachfolgern.
      ;; Ausgabe-Weg setzt diese Kette
      ;; rückwärts vom Ziel_Knoten zum
      ;; Start_Knoten als Liste zusammen.
      (Ausgabe-Weg
       (lambda (Knoten)
         (cond((null? Knoten) (list))
              (#t (append
```

```
                    (Ausgabe-Weg
                      (getprop Knoten 'Weg))
                    (list Knoten))))))
      ;;Erläuterung siehe Tabelle
      (Tiefensuche
       (lambda ()
         (cond((null? Liste-eroeffnete-Knoten)
                    (list))
               (#t (set! Analyse-Knoten  ; vgl. A1.1
                         (car Liste-eroeffnete-Knoten))
                   (set! Liste-eroeffnete-Knoten
                         (cdr Liste-eroeffnete-Knoten))
                   (set! Liste-geschlossene-Knoten
                         (cons Analyse-Knoten
                               Liste-geschlossene-Knoten))
                   (cond((eq? Analyse-Knoten Ziel_Knoten)
                          (Ausgabe-Weg Analyse-Knoten))
                          (#t (set! Arbeitsliste
                                    (Get-Nachfolger
                                      Analyse-Knoten))
                              (set! Arbeitsliste
                                    (Differenz-Knoten
                                      Arbeitsliste
                                   Liste-geschlossene-Knoten))
                              (set! Liste-eroeffnete-Knoten
                                    (append
                                       Arbeitsliste
                                       (Differenz-Knoten
                                      Liste-eroeffnete-Knoten
                                           Arbeitsliste)))
                              (for-each
                                  (lambda (Knoten)
                                     (putprop
                                        Knoten
                                        'Weg
                                         Analyse-Knoten))
                                   Arbeitsliste)
                              (Tiefensuche)))))))
    ;;Um ein Abbruchkriterium
    ;; für Ausgabe-Weg zu haben.
    (putprop Start_Knoten 'Weg (list))
    (Tiefensuche))))
eval> (Depth-first-Search 'H 'F)
      ==> (H C A B E F)
```

Das Programm `Depth-first-Search` expandiert stets den ersten Knoten von `Liste-eroeffnete-Knoten`. Anders formuliert: Die Nachfolger von `(car Liste-eroeffnete-Knoten)` bilden die Arbeitsliste. Ändern wir z. B. für den Knoten `H` die Reihenfolge der Nennungen bei der Eigenschaft `Nachfolger` (↪ Programm in Tabelle 2.5 S. 199), dann erhalten wir für unser Beispiel mit Start bei `H` und Ziel bei `F` einen kürzeren Weg; statt 35 nur 19 Entfernungseinheiten.

```
eval> (putprop 'H 'Nachfolger '((G 17.5) (C 14)))
```

```
eval> (Depth-first-Search 'H 'F) ==> (H G F)
```

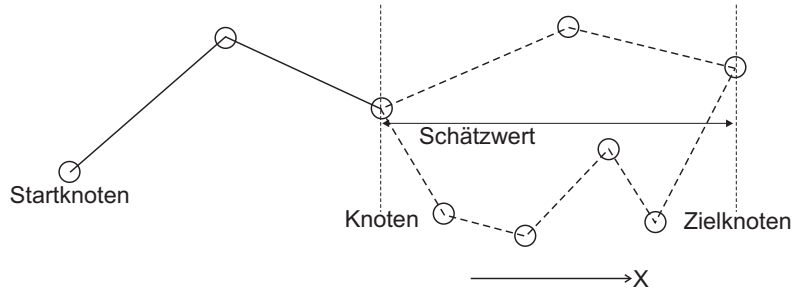
Ob die Tiefensuche für eine vorgegebene Start-Ziel-Angabe einen kurzen Weg ermittelt, hängt von der Reihenfolge der Knoten in der `Liste-eroeffnete-Knoten` ab, die sich aufgrund der Reihenfolge der Nachfolgeknoten bestimmt. Die Tiefensuche ermittelt stets einen Weg vom Startknoten zum Zielknoten, falls ein solcher existiert, aber nur „zufällig“ ist dieser Weg der kürzeste Weg. Wir können es nicht der zufällig passenden Reihenfolge in der `Liste-eroeffnete-Knoten` überlassen, welchen Knoten wir expandieren, wenn das Suchverfahren den kürzesten Weg ermitteln soll; natürlich ohne zunächst alle Knoten zu expandieren.

Wir benötigen für die Auswahl des zu expandierenden Knotens eine zusätzliche Information. Diese Angabe muss es gestatten, eine Priorität für das Expandieren eines Knotens zu ermitteln, so dass das bisherige „blinde“ Suchen ersetzt werden kann durch ein prioritäts-gesteuertes Suchen; z. B. nach dem A^*-Verfahren (↪ Programm S. 207). Benötigt wird eine sogenannte „Evaluationsfunktion“ (engl.: *evaluation function*) für das Expandieren der Knoten. (Achtung! Nicht verwechseln mit dem `eval`-Konstrukt.).

Die Priorität ergibt sich aufgrund einer Entfernungsschätzung vom Start zum Ziel bei dem Weg über den jeweiligen Knoten. Den zurückgelegten Weg vom Start bis zum Knoten merken wir uns in der P-Liste des Knotens unter der Eigenschaft `Entfernung-vom-Start`. Addieren wir zu diesem Wert eine Schätzung vom Knoten zum Ziel, dann haben wir einen Schätzwert für die Gesamtentfernung, die wir unter der Eigenschaft `Schaetzung-Gesamtentfernung` speichern. Wird die `Liste-eroeffnete-Knoten` nach den Werten von `Schaetzung-Gesamtentfernung` sortiert, dann ist ihr erster Knoten die beste Wahl, weil er den kürzesten Gesamtweg verspricht.

Das Problem liegt im Berechnen eines geeigneten Schätzwertes. Auf keinen Fall darf der Schätzwert größer sein als die tatsächliche Entfernung zwischen dem Knoten und dem Ziel (Zulässigkeitsbedingung für das A^*-Verfahren). Andernfalls wäre nicht sichergestellt, dass der erste Knoten von `Liste-eroeffnete-Knoten` die höchste Priorität hat. Es könnte dann ein Knoten mit höherem Schätzwert letztlich eine kürzere Entfernung repräsentieren als der vermeintlich beste Knoten.

Nehmen wir zur Erfüllung dieser Bedingung als extremen Schätzwert stets 0 an, dann entspricht das A^*-Verfahren einem uninformierten Verfahren, mit der Konsequenz, dass wir viele Knoten expandieren müssen. Ein geeigneter Schätzwert ist die Differenz der x-Koordinatenwerte (↪ Abbildung 2.14 S. 206). Wie immer der Weg vom Knoten zum Zielknoten auch verläuft, kürzer als die Differenz der x-Koordinatenwerte kann er nicht sein, höchstens gleich groß. Damit wird auch die Konsistenzbedingung erfüllt. Sie fordert, dass für zwei beliebige Knoten K_i und



Legende:
— ≡ zurückgelegter Weg
---- ≡ mögliche Wege

Abbildung 2.14: Differenz der X-Koordinatenwerte als Schätzung für die kürzeste Entfernung vom Knoten zum Zielknoten

K_j die Differenz der Schätzwerte nicht größer ist als die Differenz der tatsächlichen Entfernungen zum Zielknoten.

Zum Verstehen der Aufgabe sind vorab Anforderungen, Annahmen zum Entwurf und Testfälle formuliert. Dabei verweist die Abkürzung `A` auf eine *Anforderung*, die Abkürzung `E` auf eine Annahme zum *Entwurf* und die Abkürzung `T` auf einen *Testfall*.

`A1`: `A*-Search` ermittelt in einem angegebenen Knotennetz den kürzesten Weg von einem Start- zu einem Ziel-Knoten.

- `A1.1`: Der Nachfolger mit dem kleinsten Wert der Funktion `Entfernungsschaetzung-Start->Ziel` (↪ `E2`) wird als nächster analysiert (expandiert). Zweck: Der „beste“ Knoten zuerst.
- `A1.2`: Es sind ein Start- und ein Zielknoten anzugeben.

`E1`: Ein Knoten ist ein Symbol.

- `E1.1`: Die Koordinaten eines Knoten sind als Eigenschaft `Ort` gespeichert.
- `E1.2`: Die Nachfolger eines Knotens und ihre Entfernungen sind als Eigenschaft `Nachfolger` gespeichert und zwar als folgende A-Liste: $(\{(< nachfolger >< entfernung >)\})$

`E2`: `Entfernungsschaetzung-Start->Ziel` addiert zum zurückgelegten Weg vom Start zum Knoten den Absolutbetrag der Differenz der x-Koordinatenwerte zwischen Knoten und Zielknoten (↪ Abbildung 2.14 S. 206).

E3: Zwischen zwei nicht miteinander verbundenen Knoten ist die Entfernung unendlich; hier ist es der Wert 9.9*e*99.

T1: Beispiel

T1.1: Daten ↪ Programm in Tabelle 2.5 S. 199.

T1.2: Applikation (↪ Tabelle 2.7 S. 212)

```
eval> (A*-Search 'U 'H) ==> (U E C H)
```

Programm: A*-Search

```
(define A*-Search (lambda (Start_Knoten Ziel_Knoten)
  (letrec
     ((Liste-eroeffnete-Knoten (list Start_Knoten))
      (Liste-geschlossene-Knoten (list))
      (Analyse-Knoten (list))
      (Arbeitsliste (list))

      (Get-Nachfolger
       (lambda (Knoten)
         (map (lambda (x)(car x))
              (getprop Knoten 'Nachfolger))))

      (Entfernungsschaetzung-Start->Ziel
       (lambda (Knoten)
         (+ (getprop Knoten 'Entfernung-vom-Start)
            (Entfernungsschaetzung-Knoten->Ziel
              Knoten))))

      (Entfernungsschaetzung-Knoten->Ziel
       (lambda (Knoten)
         (abs (- (car (getprop Ziel_Knoten 'Ort))
                 (car (getprop Knoten 'Ort))))))

      (Entfernung-Start-Knoten->Anderer-Knoten
       (lambda (Knoten Anderer_Knoten)
         (+ (getprop Knoten 'Entfernung-vom-Start)
            ;;Falls Knoten nicht mit Anderer_Knoten
            ;; verbunden ist, wird als
            ;; Pseudoentfernung ein großer Wert
            ;; angenommen (vgl. E3).
            (cond((cadr (assoc
                           Anderer_Knoten
                           (getprop Knoten
                                    'Nachfolger))))
                 (#t 9.9e99)))))

      ;;Knoten in Liste-eroeffnete-Knoten
      ;; sind in der Reihenfolge ihrer
      ;; Prioritaet sortiert. Zugaenge
      ;; sind daher entsprechend ihres
```

```
;; Schaetzwertes einzufuegen.
(Get-beste-Knoten (lambda (Liste)
                    (car Liste)))

(Einfuegen-nach-Entfernungsschaetzung
 (lambda (Knoten Liste)
   (letrec
      ((Neue-Wert
        (Entfernungsschaetzung-Start->Ziel
          Knoten))
       (Einfuegen
        (lambda (K L)
          (cond((null? L) (list K))
                ((< Neue-Wert
                    (getprop (car L)
               'Schaetzung-Gesamtentfernung))
                  (cons K L))
                 (#t (cons (car L)
                    (Einfuegen K (cdr L))))))))
     (Einfuegen Knoten Liste))))

;;Offene-Knoten schreibt die
;; Eigenschaftsliste für die
;; neuen Knoten der bereinigten
;; Arbeitsliste fort und
;; sortiert diese in
;; Liste-eroeffnete-Knoten ein.
(Offene-Knoten
 (lambda (Knoten)
   (putprop Knoten
       'Entfernung-vom-Start
       (Entfernung-Start-Knoten->Anderer-Knoten
         Analyse-Knoten Knoten))
   (putprop Knoten
       'Schaetzung-Gesamtentfernung
       (Entfernungsschaetzung-Start->Ziel
         Knoten))
   (putprop Knoten 'Weg Analyse-Knoten)
   (set! Liste-eroeffnete-Knoten
         (Einfuegen-nach-Entfernungsschaetzung
            Knoten
            Liste-eroeffnete-Knoten))))

;;Neuberechnen-offene-Knoten ist
;; erforderlich für die Knoten,
;; die auf einem anderen Weg schon
;; erreicht wurden, d.h. sich in
;; Liste-eroeffnete-Knoten und
;; Arbeitsliste befinden.
(Neuberechnen-offene-Knoten
 (lambda (Knoten)
   (letrec
      ((Neue-Wert
        (Entfernung-Start-Knoten->Anderer-Knoten
           Analyse-Knoten Knoten)))
     (cond((< Neue-Wert
```

```
                (getprop Knoten
                         'Entfernung-vom-Start))
              (putprop Knoten
                 'Entfernung-vom-Start
                  Neue-Wert)
              (putprop Knoten
                 'Schaetzung-Gesamtentfernung
                 (Entfernungsschaetzung-Start->Ziel
                   Knoten))
              (putprop Knoten 'Weg Analyse-Knoten)
              (set! Liste-eroeffnete-Knoten
                (Einfuegen-nach-Entfernungsschaetzung
                       Knoten
                       (delete! Knoten
                         Liste-eroeffnete-Knoten))))
            (#t #f)))))

;;Gemeinsame-Knoten gibt eine Liste zurück,
;; die nur Knoten enthält, die in
;; Liste_1 und Liste_2 vorkommen.
(Gemeinsame-Knoten
 (lambda (Liste_1 Liste_2)
   (cond((null? Liste_1) null)
         ((member (car Liste_1) Liste_2)
          (cons (car Liste_1)
                (Gemeinsame-Knoten
                  (cdr Liste_1) Liste_2)))
         (#t (Gemeinsame-Knoten
               (cdr Liste_1) Liste_2)))))

;;Differenz-Knoten gibt Liste_1
;; zurueck ohne die Elemente,
;; die in Liste_2 vorkommen.
(Differenz-Knoten
 (lambda (Liste_1 Liste_2)
   (cond((null? Liste_1) null)
         ((member (car Liste_1) Liste_2)
          (Differenz-Knoten
            (cdr Liste_1) Liste_2))
         (#t (cons (car Liste_1)
                   (Differenz-Knoten
                     (cdr Liste_1) Liste_2))))))


;;Die Eigenschaft Weg verknuepft den
;; Analyse-Knoten mit seinen Nachfolgern.
;; Ausgabe-Weg setzt diese Kette rueck-
;; waerts von Ziel_Knoten zum Start_Knoten
;; als Liste zusammen.
(Ausgabe-Weg
 (lambda (Knoten)
   (cond((null? Knoten) null)
         (#t (append
               (Ausgabe-Weg
                 (getprop Knoten 'Weg))
               (list Knoten))))))
```

```
  ;;Loescht den Knoten aus der Liste
  (delete!
   (lambda (Knoten Liste)
    (cond ((null? Liste) (list))
          ((equal? Knoten (car Liste))
            (delete! Knoten (cdr Liste)))
          (#t (cons (car Liste)
                (delete! Knoten
                        (cdr Liste)))))))

  ;;Erlaeuterung siehe Tabelle
  (A*-Suche
   (lambda ()
     (cond((null? Liste-eroeffnete-Knoten)
                 null)
           (#t (set! Analyse-Knoten
                     (Get-beste-Knoten
                       Liste-eroeffnete-Knoten))
              (set! Liste-eroeffnete-Knoten
                     (delete! Analyse-Knoten
                        Liste-eroeffnete-Knoten))
              (set! Liste-geschlossene-Knoten
                     (cons Analyse-Knoten
                       Liste-geschlossene-Knoten))
              (cond((eq? Analyse-Knoten
                         Ziel_Knoten)
                    (Ausgabe-Weg Analyse-Knoten))
                   (#t (set! Arbeitsliste
                             (Get-Nachfolger
                               Analyse-Knoten))
                      (for-each Offene-Knoten
                         (Differenz-Knoten
                            (Differenz-Knoten
                               Arbeitsliste
                               Liste-geschlossene-Knoten)
                            Liste-eroeffnete-Knoten))
                      (for-each
                        Neuberechnen-offene-Knoten
                         (Gemeinsame-Knoten
                            Arbeitsliste
                            Liste-eroeffnete-Knoten))
                      (A*-Suche)))))))))

;;Um Abbruchkriterium
;; fuer Ausgabe-Weg zu haben.
(putprop Start_Knoten 'Weg (list))
;;Initialisierung
(putprop Start_Knoten 'Entfernung-vom-Start 0)
(putprop Start_Knoten
            'Schaetzung-Gesamtentfernung
            (Entfernungsschaetzung-Start->Ziel
              Start_Knoten))

;;Start der Hauptfunktion
(A*-Suche))))
```

Da das A^*-Verfahren den kürzesten Weg ermittelt, können wir Ziel- und Start-Knoten vertauschen und erhalten den gleichen Weg, wenn wir die erzeugte Liste umkehren. Bei der Tiefensuche ist dies nicht der Fall.

```
eval> (Depth-first-Search 'U 'H)
      ==> (U E C H)
eval> (reverse (Depth-first-Search 'H 'U))
      ==> (U E B A C H)
eval> (A*-Search 'U 'H)
       ==> (U E C H)
eval> (reverse (A*-Search 'H 'U))
      ==> (U E C H)
```

2.2.4 Zusammenfassung: Liste, A-Liste und P-Liste

Die Assoziationsliste (kurz: A-Liste) ist eine Liste, deren Elemente Sublisten sind, wobei das erste Element jeder Subliste die Aufgabe eines Zugriffsschlüssels hat. Statt Sublisten können auch Punkt-Paare stehen. Genutzt werden Konstruktionsmittel für die „einfache“ Liste, d. h. die Konstruktoren `cons` und `list`, die Selektoren `car`, `cdr` und `list-ref`, das Prädikat `pair?` und die Mutatoren `set!`, `set-car!` und `set--cdr!`. Der Mutator `set!` dient zum Benennen einer existierenden (Teil-)Liste. Die Mutatoren `set-car!` und `set-cdr!` ändern Listenelemente. Die Selektoren für ein A-Listen-Element (Subliste) prüfen das Schlüsselelement. Der Selektor `assoc` vergleicht mit `equal?` und der Selektor `assq` mit `eq?`.

In klassischen LIPS-Systemen ist jedem Symbol eine zunächst leere Eigenschaftsliste (engl.: *property list*; kurz: P-Liste) zugeordnet. Das `putprop`-Konstrukt fügt eine neue Eigenschaft ein bzw. überschreibt eine schon vorhandene. Das `getprop`-Konstrukt selektiert den Wert einer angegebenen Eigenschaft. Die P-Liste ist global. Dies hat Vor- und Nachteile. Vorteilhaft kann die P-Liste zur Kommunikation zwischen Konstrukten genutzt werden. Nachteilig ist die Gefahr ungewollter Nebeneffekte.

Charakteristische Beispiele für Abschnitt 2.2

Zirkuläres Punkt-Paar

```
eval> ;;;;Zirkuläres Punkt-Paar
      ;;;; Genutzte Bibliotheken von Scheme R6RS
      (require rnrs/base-6)
eval> (require rnrs/mutable-pairs-6)
eval> (define Endlos (cons 'A 'B))
eval> (set-cdr! Endlos Endlos)
```

```
eval> (A*-Search 'H 'U) ==> (H C E U)
```

Hausposststation	x-Koordinate	y-Koordinate
A	8.5	11.5
B	15.5	10.0
C	8.5	6.5
D	8.5	5.5
E	14.0	6.5
F	15.5	4.5
G	15.5	3.0
H	0.5	0.5
U	17.0	6.5

Applikation von Offene-Knoten auf den Knoten	P-Liste des Knotens (ohne Ort) Weg	Schaetzung-Gesamtentfernung	Entfernung vom Start	Nachfolger	Liste-er-oeffnete-Knoten
C	H	22.5	14.0	((D 1) (A 5) (E 5.5) (H 14))	(C)
G	H	19.0	17.5	((F 1.5) (H 17.5))	(G C)
F	G	20.5	19.0	((E) (G 1.5))	(F C)
E	F	25.0	22.0	((C 5.5) (F) (B 4.5) (U 3))	(E C)
D	C	2.5	15.0	((C 1))	(D E)
A	C	27.5	19.0	((C 5) (B 8.5))	(A D E)
Achtung! Neuberechnen-offene-Knoten für E:					
E	C	22.5	19.5	((C 5.5) (F) (B 4.5) (U 3))	(E A D)
B	E	25.5	24.0	((A 8.5) (E 4.5))	(B A D)
U	E	22.5	22.5	((E))	(U B A D)

; U = Ziel_Knoten

Legende:
Büro (Daten) ↪ Programm in Tabelle 2.5 S. 199
A^*-Programm ↪ S. 207.

Tabelle 2.7: Applikation des Offene-Knoten-Konstruktes in $A^\star$ bei Startknoten H, Zielknoten U

```
eval> (set-car! Endlos
        "LISP: Das ultimative Modell!")
eval> Endlos
      #0={"LISP: Das ultimative Modell!" . #0#}
eval> (list-ref Endlos 100) ==>
      "LISP: Das ultimative Modell!"
```

Zweidimensionale Matrix

```
;;;;Zweidimensionale Matrix konstruiert
;;;; als A-Liste.

;;;Mit Hilfe von Mutatoren
;;; wird die lokale Matrix fortgeschrieben.
;;; Ein Element wird mit dem assoc-Konstrukt
;;; selektiert.
;;; Genutzte Sprache: Scheme R5RS
eval> (define Make-x-y-Speicher
       (lambda ()
        (let* ((Matrix (list '*MATRIX*))
               ;;Selektor für ein
               ;; Matrixelement
               (get (lambda (x y)
                 (let ((Zeile (assoc x
                                (cdr Matrix))))
                   (cond((not(pair? Zeile))
                         (list))
                        (#t (let ((Spalte
                                   (assoc y
                                    (cdr Zeile))))
                              (cond((not(pair?
                                     Spalte))
                                    (list))
                                   (#t (cdr
                                         Spalte)
                                   ))))))))
               ;;Konstruktor/Mutator für
               ;; ein Matrixelement
               (put! (lambda (x y Wert)
                 (let ((Zeile (assoc x
                                (cdr Matrix))))
                   (cond((not(pair? Zeile))
                             (set-cdr! Matrix
                               (cons (list x
                                       (cons
                                         y Wert))
                                     (cdr Matrix))))
                         (#t  (let ((Spalte
                                     (assoc y
```

```
                                        (cdr Zeile)
                                       )))
                                  (cond((not(pair?
                                         Spalte))
                                         (set-cdr!
                                           Zeile
                                          (cons
                                            (cons y Wert)
                                            (cdr Zeile)
                                           )))
                                        (#t (set-cdr!
                                              Spalte
                                              Wert)))))))
                    Matrix))
                ;;Verzweigungsfunktion
                ;; zum Selektieren der
                ;; jeweiligen Operation
                (Dispatch
                  (lambda (op)
                    (cond((eq? op 'get) get)
                          ((eq? op 'put!) put!)
                          (#t
                            (display
                             "Unbekannte Operation: ")
                             op)))))
              ;;Rückgabewert ist die
              ;; Verzweigungsfunktion
              Dispatch)))

eval> (define x-y-Matrix (Make-x-y-Speicher))
eval> x-y-Matrix ==> #<procedure:dispatch>
eval> (define put! (x-y-matrix 'put!))
eval> put! ==> #<procedure:put!>
eval> (define get (x-y-matrix 'get))
eval> (put! 1 2 "heute")
      ==> (*matrix* (1 (2 . "heute")))
eval> (get 1 2) ==> "heute"
eval> (put! 3 4 "gestern") ==>
      (*matrix* (3 (4 . "gestern"))
        (1 (2 . "heute")))
eval> (get 3 4) ==> "gestern"
eval> (put! '(a b c) #\a) ==> ERROR ...
      ;procedure put!: expects 3
      ; arguments, given 2: (a b c) #\a
eval> (put! '(a b c) #\a "Alles klar?") ==>
      (*matrix* ((a b c) (#\a . "Alles klar?"))
      (3 (4 . "gestern"))
      (1 (2 . "heute")))
eval> (get '(a b c) #\a) ==> "Alles klar?"
```

Geschachtelte `getprop`-Konstrukte

```
;;;;Geschachtelte getprop-Konstrukte

;;;Konten mit unterschiedlichen
;;; Trend-Funktionen.
;;; Für ein Konto,z.B. K001, wird über
;;; die Eigenschaft Konto-TYP ein Symbol
;;; selektiert. Dieses Symbol hat als
;;; Wert seine Eigenschaft Trend,
;;; die anzuwendende Trend-Funktion.
;;; Das Argument für diese Funktion ist
;;; unter der Eigenschaft Buchungen
;;; gespeichert.
;;; Genutzte Sprache: Scheme mit
;;; module compat mzscheme.
eval> (putprop 'K001 'Konto-Typ 'Vermoegen)
eval> (putprop 'K001 'Buchungen '(20 30))
eval> (putprop 'K002 'Konto-Typ 'Verbrauch)
eval> (putprop 'K002 'Buchungen '(10 30 100))
eval> (putprop 'Vermoegen 'Trend
               (lambda (Konto)
                 (map (lambda (x)
                        (* 1.1 x))
                      (getprop Konto
                        'Buchungen))))
eval> (putprop 'Verbrauch 'Trend
               (lambda (Konto)
                 (map (lambda (x)
                        (+ 100 x))
                      (getprop Konto
                        'Buchungen))))
eval> (define Trend
        (lambda (Konto)
          ((getprop
            (getprop Konto 'Konto-Typ)
              'Trend)
           Konto)))
eval> (Trend 'K001) ==> (22.0 33.0)
eval> (Trend 'K002) ==> (110 130 200)
```

2.3 Abbildungsoption: Vektor

LISP-Systeme ermöglichen neben der Definition von Listen auch die Definition von Vektoren. Dabei ist ein Vektor eine modifizierbare Struktur. Den Elementen dieser Strukur ist ein Index vom Typ ganzzahlige, positive Zahl (Integer-Zahl) zugeordnet. Die Vektorelemente können von

beliebigem Typ sein, also auch wieder Vektoren. Z. B. ist ein zweidimensionales Feld (engl.: *array*) ein Vektor mit Vektoren als Elemente. Die Länge eines Vektors ist gegeben durch die Anzahl der Elemente, die er enthält. Sie wird zum Zeitpunkt der Vektordefinition festgelegt.

Im Gegensatz zur Liste ist der Vektor daher eine „statische“ Abbildungsoption. Ein Vektor hat eine signifikant andere Zugriffszeit-Charakteristik als eine Liste. Die Zugriffszeit ist konstant, d. h. sie ist unabhägig davon, ob auf ein Element am Anfang (kleiner Index) oder am Ende des Vektors (großer Index) zugegriffen wird. Bei der Liste steigt die Zugriffszeit linear mit der Position des gesuchten Elementes, wenn sie klassisch implementiert ist (↪ Abschnitt A S. 457).

Zunächst erörtern wir eingebaute Vektor-Konstrukte im Sinne des Verbundes von Konstruktor, Selektor, Prädikat und Mutator (↪ Abschnitt 2.3.1 S. 216). Mit diesen Konstrukten implementieren wir einen solchen Verbund für die Liste, d. h. wir bilden die `cons`-Zelle als Vektor ab (↪ Abschnitt 2.3.2 S. 219). Anschließend nutzen wir die Abbildungsoption Vektor, um eine Aktenverwaltung als einen höhenbalancierten Baum abzubilden (↪ Abschnitt 2.3.3 S. 222). Dieses größere Beispiel möge gleichzeitig dem Training im Lesen fremder Programme dienen.

2.3.1 Vektor-Konstrukte

In Scheme unterscheidet sich die Notation eines Vektors von der einer Liste durch ein vorrangestelltes #-Zeichen.

```
eval> #(A B C D E)       ;Vektor mit 5 Elementen
      ==> #(A B C D E)
eval> #(#(A B) #(C D) E) ;Vektor mit 3 Elementen
           ==> #(#(A B) #(C D) E)
eval> (vector-length #(#(A) (B C D E))) ==> 2
```

Wie die Beispiele zeigen, gilt für Vektoren die EVAL-Regel 1 (↪ Abschnitt 1.1.2 S. 21). Sie haben sich selbst als Wert. Wir können daher nicht (— wie im Fall der Liste —) direkt eine Funktionsapplikation notieren. Enthält der Vektor eine Funktion und ihre Argumente und ist diese Funktion anzuwenden, dann ist der Vektor vorab in eine Liste zu überführen. Dazu dient das Konstrukt `vector->list`. Das `list->vector`-Konstrukt führt die umgekehrte Transformation aus.

```
eval> (define Foo #(+ 1 2 3))
eval> Foo ==> #(+ 1 2 3)
eval> (vector->list Foo) ==> (+ 1 2 3)
eval> (eval (vector->list Foo)) ==> 6
eval> (list->vector '(A B C)) ==> #(A B C)
```

Konstruktoren:

`eval>` `(vector` $<el_0><el_1>\ldots<el_j>\ldots<el_n>$`)` `==>`
`#(`$<wert_0><wert_1>\ldots<wert_j>\ldots<wert_n>$`)`
`eval>` `(make-vector k)` `==>` `#(...)`
`eval>` `(make-vector k` $<init-sexpr>$`)`
`==>` `#(`$<value>\ldots<value>$`)`

mit:

$<el_j>$	$\equiv$ j-te Element des Vektors.
$<wert_j>$	$\equiv$ Wert von $<el_j>$, `vector` und `make-vector` evaluieren ihre Argumente.
`#(...)`	$\equiv$ Notation für einen Vektor.
k	$\equiv$ Anzahl der Elemente.
$<init-sexpr>$	$\equiv$ Initialausdruck für die generierten Elemente.
$<value>$	$\equiv$ Wert von $<init-sexpr>$.

Selektor:

`eval>` `(vector-ref` $<vektor>$ i`)` `==>` $<wert_i>$

mit:

i $\equiv 0,1,\ldots,$ `(-` `(vector-length` $<vektor>$`)` `1)`
Das erste Element hat den Index 0. Das letzte Element hat den Index der um 1 verminderten Vektorlänge. `vector-ref` ist *„zero based“*.

Prädikat:

`eval>` `(vector?` $<sexpr>$`)`
`==>` `#t` ; Falls ein $<sexpr>$ ein Vektor ist, sonst `#f`.

Mutator:

`eval>` `(vector-set!` $<vektor>$ i $<sexpr>$`)`
`==>` `#(...`$<neue-wert_i>\ldots$`)`

mit:

$<neue-wert_i>$ $\equiv$ Das Element i hat den Wert von $<sexpr>$.

Die folgenden Beispiele verdeutlichen diese `vector`-Konstrukte.

```
eval> (define *V001*
       (vector "Reppenstedt" 21391 "04131/63845"))
eval> (vector? *V001*) ==> #t
eval> (vector-ref *V001* 1) ==> 21391
eval> (vector-ref *V001* 3) ==> ERROR ...
      ;vector-ref: index 3 out of range [0, 2]
      ; for vector:
```

```
      ; #("Reppenstedt" 21391 "04131/63845")
eval> (vector-set! *V001* 2 "01626989027")
eval> *V001* ==>
      #("Reppenstedt" 21391 "01626989027")
eval> (make-vector 4 "LG") ==>
      #("LG" "LG" "LG" "LG")
eval> (vector? (make-vector 30)) ==> #t
eval> (vector? (list 1 2 3)) ==> #f
```

Destruktives Konstrukt `Modul-V!` **auf Basis von** `vector-set!`

```
eval> (define Modul-V!
        (lambda (Objekt Index Neuer_Wert)
          (cond ((and
                   (vector? Objekt)
                   (integer? Index)
                   (<= 0 Index)
                   (< Index (vector-length Objekt)))
                  (vector-set! Objekt
                      Index Neuer_Wert)
                      Objekt)
                (#t (display
                      "Vektor-Angaben fehlerhaft!")
                    Objekt))))
eva> (define *V001*
       (vector
         "Reppenstedt"
         21391
         "04131/63845"))
eval> (Modul-V!
        (Modul-V! *V001* 1 21339)
        0
        "Lüneburg")
      ==> #("Lüneburg" 21339 "04131/63845")
eval> (Modul-V! '(A B) 0 'C) ==>
       Vektor-Angaben fehlerhaft!(A B)
```

Zum Vergleich ist für `Modul-V!` **die Listenlösung** `Modul-L!` **angegeben:**

```
eval> (define Modul-L!
        (lambda (Objekt Index Neuer_Wert)
          (cond ((and
                   (pair? Objekt)
                   (integer? Index)
                   (<= 0 Index)
                   (< Index (length Objekt)))
                  (set-car! (member
                              (list-ref Objekt Index)
```

```
                              Objekt)
                             Neuer_Wert)
                  Objekt)
                 (#t (display
                       "Listenangaben fehlerhaft!")
                     Objekt))))
eval> (define *L001*
        (list "Reppenstedt" 21391 "04131/63845"))
eval> (Modul-L!
        (Modul-L! *L001* 1 21339)
        0
        "Lüneburg")
      ==> ("Lüneburg" 21339 "04131/63845")
```

Wann eine A-Liste effizienter als ein Vektor ist, d. h. eine kleinere durchschnittliche Selektionszeit hat, ist abhängig von der jeweiligen LISP-Implementation. Als Faustregel gilt, dass ein dünn besetzter Vektor besser als eine A-Liste zu implementieren ist (↪ Zweidimensionale Matrix; Abschnitt 2.2.4 S. 213).

2.3.2 Abbildung der `cons`-Zelle als Vektor

Die CONS-Zelle verknüpft zwei symbolische Ausdrücke (↪ Abschnitt 1.2.1 S. 52). Sie ist das klassische Mittel um eine Liste zu konstruieren. Wir bilden im folgenden eine `cons`-Zelle als Vektor ab. Den Konstruktor für diese `cons`-Zelle nennen wir `V-cons`. Damit vermeiden wir Namensgleichheit und infolgedessen das Überschreiben des eingebauten `cons`-Konstruktes. Selektoren, Prädikate und Mutatoren haben aus gleichem Grund den Präfix „`V-`“.

Zum Konstruieren einer Liste gehen wir von `null`, der leeren Liste, aus. Wir definieren daher ein *V-NULL* mit dem Konstrukt `vector`. Die Sternchen im Namen sollen ein versehentliches Überschreiben dieser globalen Variablen verhindern (Näheres zur Benennung ↪ Abschnitt 3.1.1 S. 377). Das `*V-NULL*` kann, wie folgt, definiert werden:

```
eval> (define *V-NULL* (vector #\0 #\0))
eval> *V-NULL* ==> #(#\0 #\0)
```

Das zugehörende Prädikat `V-null?` könnten wir so definieren, dass jeder Vektor mit den beiden Elementen `#\0` ein `*V-NULL*` repräsentiert. Das `V-null?`-Prädikat stützt sich damit auf den Vergleich der Elemente:

```
eval> (define V-null?
        (lambda (V_Pair)
          (and (vector? V_Pair)
               (= (vector-length V_Pair) 2)
               (eq? (vector-ref V_Pair 0) #\0)
               (eq? (vector-ref V_Pair 1) #\0))))
eval> (V-null? *V-NULL*) ==> #t
eval> (V-null? #(A B)) ==> #f
```

Wir unterstellen jedoch, dass es nur ein Objekt gibt, das `*V-NULL*` repräsentiert. Das Prädikat `V-null?` testet daher auf Identität und nicht auf Gleichheit von Strukturen (↪ Abschnitt 1.2.1 S. 52). Das `V-null?`--Konstrukt nutzt daher für den direkten Vektor-Vergleich das `eq?`-Konstrukt.

```
eval> (define V-null?
        (lambda (Objekt)
          (eq? Objekt *V-NULL*)))
eval> (V-null? *V-NULL*) ==> #t
eval> (V-null? #(#\0 #\0)) ==> #f
```

Der Konstruktor `cons` generiert eine neue `cons`-Zelle, um den `car`-Teil mit dem `cdr`-Teil zu verknüpfen. Diese Verknüpfung realisieren wir durch das Konstruieren eines zweielementigen Vektors.

```
eval> (define V-cons
        (lambda (car_Teil cdr_Teil)
          (vector car_Teil cdr_Teil)))
```

Die Selektoren `V-car` und `V-cdr` realisieren wir dann mit dem Selektor `vector-ref`.

```
eval> (define V-car
        (lambda (V_Pair)
          (vector-ref V_Pair 0)))
eval> (define V-cdr
        (lambda (V_Pair)
          (vector-ref V_Pair 1)))
```

Das erforderliche Prädikat `V-pair?` nutzt das eingebaute Prädikat `vector?`. Das `V-pair?`-Konstrukt testet, ob eine `V-cons`-Zelle vorliegt (bzw. mehrere). Da nicht jeder Vektor vom Typ `V-cons` ist, überprüft `V-pair?` auch die Länge des Vektors. Wir unterstellen damit, dass jeder zweielementige Vektor eine V-cons-Zelle ist.

```
eval> (define V-pair?
        (lambda (V_Pair)
          (and (vector? V_Pair)
               (= (vector-length V_Pair) 2))))
```

Die Mutatoren `V-set-car!` und `V-set-cdr!` sind mit dem eingebauten Mutator `vector-set!` leicht realisierbar.

```
eval> (define V-set-car!
        (lambda (V_Pair car_Teil)
          (vector-set! V_Pair 0 car_Teil)))

eval> (define V-set-cdr!
        (lambda (V_Pair cdr_Teil)
          (vector-set! V_Pair 1 cdr_Teil)))
```

Unser Verbund von Konstruktor `V-cons`, Selektoren `V-car` und `V--cdr` sowie Prädikat `V-null?` hat die wesentliche Eigenschaft: Das Zusammengefügte kann ohne Verlust wieder getrennt werden.

```
eval> (V-car (V-cons 'X 'Y)) ==> X
eval> (V-cdr (V-cons 'X 'Y)) ==> Y
eval> (V-null? (V-car (V-cons *V-NULL* 'Y))) ==> #t
eval> (V-null? (V-cdr (V-cons 'X *V-NULL*))) ==> #t

eval> (define Foo (V-cons 'X 'Y))
eval> Foo ==> #(X Y)
```

Der Wert von `Foo` „verrät“, dass unsere `V-cons`-Zellen als Vektoren abgebildet sind. Wir definieren ein geeignetes `V-pp`-Konstrukt[11], um auch nach außen hin das gewohnte Layout einer Listen-Darstellung vermitteln zu können.

```
eval> (define V-pp
       (lambda (V_Pair)
         (letrec
             ((V-atom?
               (lambda (Objekt)
                 (not (V-pair? Objekt))))
              (Ausgabe
               (lambda (V_Pair)
                 (cond ((V-null? V_Pair)
                        (display ") "))
                       ((V-atom?
                         (V-car V_Pair))
                        (print (V-car V_Pair))
                        (cond ((V-pair?
                                (V-cdr V_Pair))
                               (cond ((V-null?
                                       (V-cdr
                                        V_Pair))
                                      (Ausgabe
                                       (V-cdr
                                        V_Pair)))
                                     (#t
                                      (display " ")
                                      (Ausgabe
                                       (V-cdr
                                        V_Pair)))))
                               (#t (display "(")
                                   (Ausgabe
                                    (V-car
                                     V_Pair))
```

[11] Abkürzung `pp` für *pretty print*. PP-Konstrukte enthält die *PLT-Scheme*-Bibliothek `scheme/pretty`, d. h. `(require scheme/pretty)` ↪ S. 473.

```
                                    (Ausgabe
                                     (V-cdr
                                       V_Pair)))
                                ))))))
           (cond ((V-pair? V_Pair)
                  (display "(")
                  (Ausgabe V_Pair))
                 (#t (print V_Pair)
                     (void))))))
eval> (define Foo (V-cons "Emma"
        (V-cons "Krause" *V-NULL*)))
eval> Foo ==>
      #("Emma" #("Krause" #(#\0 #\0)))
eval> (V-pp Foo)
      ==> ("Emma" "Krause")
eval> (define Bar (V-cons "Frau" Foo))
eval> Bar ==>
      #("Frau" #("Emma" #("Krause" #(#\0 #\0))))
eval> (V-pp Bar) ==>
      ("Frau" "Emma" "Krause")
eval> (V-pp
       (V-cons '*TAB*
               (V-cons '(KeyA 1)
                  (V-cons '(KeyB 2)
                    (V-cons '(KeyC 3)
                       *V-NULL*)))))
      ==> (*TAB* (KeyA 1) (KeyB 2) (KeyC 3))
```

2.3.3 Höhenbalancierter Baum

Beispiel: Aktenverwaltung

Als Beispiel dient eine Verwaltung von Akten, bei der man neue Akten anlegen und angelegte Aktenbeschreibungen modifizieren; jedoch keine Akten entfernen (löschen) kann (↪ Programm S. 229) Unterstellt wird für jede Akte ein eindeutiges Aktenzeichen (`Key`). Jedem Aktenzeichen kann ein beliebiger Text als Inhaltsbeschreibung zugeordnet werden. Die erfassten Akten können in einer anzugebenden Datei abgelegt und von dieser wieder eingelesen werden.

Da unterstellbar ist, dass in dieser Aktenverwaltung häufig gesucht und neue Akten eingefügt werden, speichern wir die Akten als „höhenbalancierten Baum“ (engl.: *balanced tree*). Für einen binären Baum gilt folgende Definition der Ausgeglichenheit:

Defintion: „Ausgeglichenheit“. Ein Baum ist genau dann ausgeglichen, wenn sich für jeden Knoten die Höhen der zugehörigen Teilbäume

um höchstens 1 unterscheiden.[12] Dabei ist die Höhe eines Baumes gleich der maximalen Astlänge (größte Knotenzahl zu einem Baumblatt).

Ein solcher ausgeglichener Baum wird zu Ehren von *Adelson-Velskii* und *Landis* kurz als *AVL*-Baum bezeichnet. In einigen LISP-Systemen sind Konstrukte für die Handhabung von *AVL*-Bäumen integriert (↪ Cambridge LISP (↪ Abschnitt A.2 S. 462); Näheres zu *AVL*-Bäumen ↪ z. B. [112, 134]).

Ein *AVL*-Baum ist durch einen Vektor abbildbar, der die Wurzel darstellt. Für unsere Aktenverwaltung ist es ein Vektor mit den fünf Elementen:

1. Höhe des Baums,
2. Aktenzeichen (`Key`),
3. zugeordnete Aktenbeschreibung (`Text`),
4. linker Teilbaum und
5. rechter Teilbaum.

Die sogenannte Balance *B* eines Baums ist für einen Knoten, wie folgt, definiert:

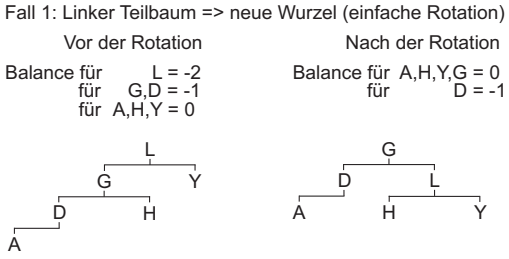
$B \equiv$ Höhe des rechten Teilbaums minus Höhe des linken Teilbaums

Zum Aufbau eines *AVL*-Baums dient ein Anfangsbaum, hier *LEERER-BAUM*. In diesen leeren Baum werden zunächst Aktenzeichen, d. h. neue Knoten, eingefügt. Beim Einfügen ist zu überprüfen, ob weiterhin das Kriterium der Ausgeglichenheit gegeben ist. Liegt die Ausgeglichenheit nicht mehr vor, dann ist abhängig vom Baumzustand eine Rotation vorzunehmen. Bei der Rotation ist der jeweilige Wurzelknoten neu zu ermitteln. Die bisherigen Teilbäume sind „umzuhängen“, so dass der linke Ast stets zu den kleineren Key-Werten und der rechte Ast zu den größeren Key-Werten führt. Bei dieser Rotation sind vier Fälle zu unterscheiden (↪ Abbildungen 2.15 S. 224 bis 2.18 S. 225). Das Einfügen eines Knotens ist als Konstruieren eines neuen *AVL*-Baums implementiert und nicht mit Hilfe des Mutators `vector-set!`. Der Grund ist hier das leichtere Verstehen des Beispielprogramms.

Das Konstrukt `Aktenverwaltung` nutzt die Möglichkeit eine `lambda`-Schnittstelle (↪ Abschnitt 1.1.4 S. 32) für eine unbestimmte Anzahl von Argumenten zu definieren.

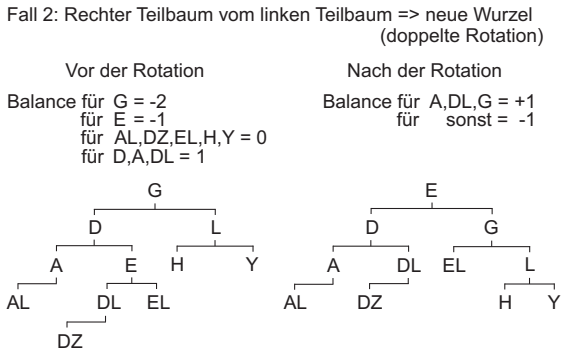
```
eval> (define Aktenverwaltung
         (lambda
           (Operation . Parameter) ;Schnittstelle
              ... )                ;Funktionskörper
```

[12] Adelson-Velskii/Landis, 1962; zitiert nach ↪ [191] S. 244.



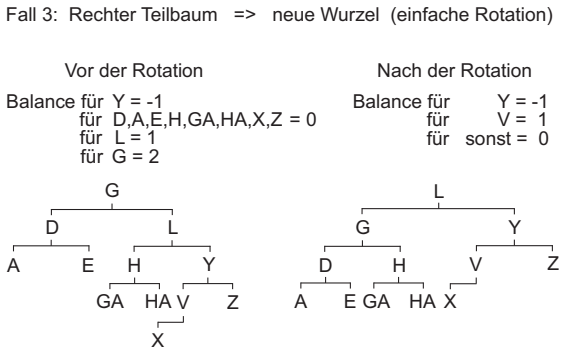
Legende:
Notation ↪ Abbildung 2.19 S. 226
Ausbalancieren eines Baums durch Rotation — Beispiel zum Verstehen des Konstruktes `AVL-Baum-Montage` (↪ Programm S. 229)

Abbildung 2.15: Baum — Fall 1: einfache Rotation



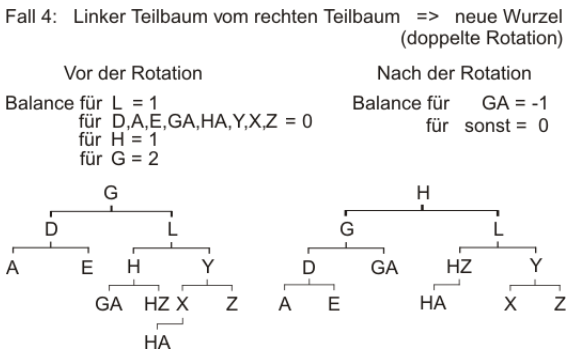
Legende:
Notation ↪ Abbildung 2.19 S. 226
Ausbalancieren eines Baums durch Rotation — Beispiel zum Verstehen des Konstruktes `AVL-Baum-Montage` (↪ Programm S. 229)

Abbildung 2.16: Baum — Fall 2: doppelte Rotation



Legende:
Notation ↪ Abbildung 2.19 S. 226
Ausbalancieren eines Baums durch Rotation — Beispiel zum Verstehen des Konstruktes `AVL-Baum-Montage` (↪ Programm S. 229)

Abbildung 2.17: Baum — Fall 3: einfache Rotation



Legende:
Notation ↪ Abbildung 2.19 S. 226
Ausbalancieren eines Baums durch Rotation — Beispiel zum Verstehen des Konstruktes `AVL-Baum-Montage` (↪ Programm S. 229)

Abbildung 2.18: Baum — Fall 4: doppelte Rotation

A, ... , Z ::= Ausprägungen des Schlüssels (Key) für den binären Baum. Sie repräsentieren einen Knoten als folgenden Vektor:

#(2 “D” “Text” <linker-Teilbaum> <rechter-Teilbaum>)

mit:
< *linker – Teilbaum* >≡
`#(1 "A" <text> *LEERER-BAUM* *LEERER-BAUM*)`

< *rechter – Teilbaum* >≡
`#(1 "E" <text> *LEERER-BAUM* *LEERER-BAUM*)`

< *text* >≡ Zum KEY zugeordnete Daten

`eval> *LEERER-BAUM* ==> #(0 "" "" () ())`

Legende:
↪ Abbildungen 2.19 S. 226 bis 2.18 S. 225

Abbildung 2.19: Baum — Notationserläuterung

Die `lambda`-Variable `Operation` wird an den Wert des ersten Arguments gebunden. Aufgrund der Angabe des Punkt-Paares (!!!) werden alle Werte von weiteren Argumenten zunächst zu einer Liste zusammengefasst, dann wird die `lambda`-Variable `Parameter` an diese Liste gebunden. Gibt es keine weiteren Argumente, hat `Parameter` den Ersatzwert `null`.

```
eval> (Aktenverwaltung 'Eingabe
        "A-12" "Fall Krause")
      ;Während der Abarbeitung
      ; dieser Applikation
      ; sind folgende Bindungen
      ; gegeben:
      ;  Operation --> Eingabe
      ;  Parameter --> ("A-12" "Fall Krause")
```

Diese `lambda`-Schnittstelle gestattet es z. B., die Aktenverwaltung zum Anzeigen der erfassten Aktenzeichen mit weniger Argumenten aufzurufen als beim Einfügen eines Aktenzeichens.

```
eval> (Aktenverwaltung 'Struktur-zeigen) ==> ...
      ;Während der Abarbeitung dieser
      ; Applikation sind folgende
      ; Bindungen gegeben:
      ;    Operation --> Struktur-zeigen
      ;    Parameter --> ()
           ;Da kein Argument hierfür angegeben ist,
           ; wird der Ersatzwert angenommen.
```

Die globale Variable `*DATEN*` hat den aktuellen Bearbeitungsstand,

d. h. den *AVL*-Baum, als Wert. Wir sichern diesen mit Hilfe des eingebauten Konstruktes `with-output-to-file`. Das Laden des gesicherten *AVL*-Baumes erfolgt mit dem Konstrukt `with-input-from-file`. Diese beiden Konstrukte arbeiten mit einem `lambda`-Konstrukt ohne `lambda`-Variable (*thunk*, ↪ Abschnitt 1.2.3 S. 103). Es wird ausgeführt mit der angegebenen Datei als aktuelle Datenquelle bzw. Datensenke.

```
eval> (define *DATEN* "... AVL-Baum ...")
eval> ;;Sicherung des AVL-Baumes
     (with-output-to-file
          "D:\\bonin\\scheme\\AVL.dat"
        (lambda ()
          (print *DATEN*)))
eval> ;;Laden des AVL-Baumes
      (set! *DATEN*
            (with-input-from-file
                "D:\\bonin\\scheme\\AVL.dat"
              (lambda ()
                (read))))
```

Zum Verstehen der Aufgabe sind vorab Anforderungen, Annahmen zum Entwurf und Testfälle formuliert. Dabei verweist die Abkürzung `A` auf eine *Anforderung*, die Abkürzung `E` auf eine Annahme zum *Entwurf* und die Abkürzung `T` auf einen *Testfall*.

`A1`: Aktenverwaltung speichert einen Aktenbestand.

- `A1.1`: Eine Akte besteht aus einem Aktenzeichen (≡ `Key`) und einem Erläuterungstext (≡ `Text`).
- `A1.2`: Ein höhenbalancierter Baum (*AVL*-Baum) repräsentiert den Aktenbestand.

`A2`: Ein neuer `Key` ist in den *AVL*-Baum einfügbar. Dabei wird geprüft, ob der Baum ausgeglichen ist, falls nicht, wird ein neuer *AVL*-Baum konstruiert.

`A3`: Ein `Text` ist durch einen neuen ersetzbar.

`A4`: Ein gespeicherter `Key` ist aus dem *AVL*-Baum nicht löschbar

`E1`: Jeder Baumknoten ist ein Vektor, der Baumhöhe, `Key`, `Text` und die beiden Teilbäume abbildet:

$$<knoten> \equiv \quad \#(<hoehe><key><text> \\ <linker-teilbaum><rechter-teilbaum>)$$

`E2`: `Key` hat den Typ Zeichenkette; ↪ [A1.1].

`E3`: `Text` hat einen beliebigen Typ; ↪ [A1.1].

E4: Zum Ausbalancieren des *AVL*-Baumes (beim Einfügen von neuen Aktenzeichen) wird eine `Key`-Gleichheit mit dem `string-ci=?`-Konstrukt festgestellt. Es gibt daher keine Unterscheidung durch eine Gross-/Kleinschreibung. Z. B. ist `"A-1-123"` damit gleich `"a-1-123"`.

E5: Benutzerschnittstelle:

E5.1: Laden eines Aktenbestandes:
`(Aktenverwaltung 'Start` $<$ *datei* $>$`)`

E5.2: Sichern eines Aktenbestandes:
`(Aktenverwaltung 'Ende` $<$ *datei* $>$`)`

E5.3: Eingeben/Modifizieren einer Akte:
`(Aktenverwaltung 'Eingabe` $<$ *key* $>$ $<$ *text* $>$`)`

E5.4: Anzeigen Akte:
`(Aktenverwaltung 'Get-text` $<$ *key* $>$`)`

E5.5: Kleinste Aktenzeichen:
`(Aktenverwaltung 'Get-min-Key)`

E5.6: Größte Aktenzeichen:
`(Aktenverwaltung 'Get-max-Key)`

E5.7: Nächstes Aktenzeichen:
`(Aktenverwaltung 'Get-next-Key)`

E5.7: Bestand an Aktenzeichen als *AVL*-Baum zeigen:
`(Aktenverwaltung 'Struktur-zeigen)`

T1: Beispiel

T1.1: Aktenbestand:

```
eval> (begin
       (Aktenverwaltung 'Eingabe
         "L-a-2" "Fall Meyer")
       (Aktenverwaltung 'Eingabe
         "L-a-2/b" "Fortsetzung Meyer")
       (Aktenverwaltung 'Eingabe
         "G-b-3-2" "Plan Westfeld")
       (Aktenverwaltung 'Eingabe
         "Y-345-6" "Fall Schulze")
       (Aktenverwaltung 'Eingabe
         "Dortmund" "Stadtplanung")
       (Aktenverwaltung 'Eingabe
         "Herbert" "Fall Herbert")
       (Aktenverwaltung 'Eingabe
         "A-1-122" "Rechnerbeschaffung")
       (Aktenverwaltung 'Eingabe
         "G-b-3-2" "Plan Neu-Westfeld")
```

```
                (Aktenverwaltung 'Ende
                  "D:\\bonin\\scheme\\AVL.DAT"))
```

T1.2: Erzeugter AVL-Baum ; <key> /<höhe>/

```
        eval> (Aktenverwaltung 'Struktur-zeigen) ==>
                   Y-345-6 /1/
                 L-a-2/b /2/
               L-a-2 /4/
                   Herbert /1/
                 G-b-3-2 /3/
                   Dortmund /2/
                     A-1-122 /1/
```

Programm: Aktenverwaltung

```
(define Aktenverwaltung
 (lambda (operation . parameter)
  (letrec
;;;;Konstruktoren, Selektoren und Prädikate
     ((Make-Baum
       (lambda (Hoehe Key Text
                linker-Teil rechter-Teil)
         (vector Hoehe Key Text
                linker-Teil rechter-Teil)))
      (AVL-Baum?
       (lambda (Baum)
         (cond((leerer-Baum? Baum) Baum)
              ((and (vector? Baum)
                    (= (vector-length Baum) 5)
                    (integer? (Baum-Hoehe Baum))
                    (positive? (Baum-Hoehe Baum))
                    (string? (Baum-Key Baum))
                    (AVL-Baum?
                      (linker-Teilbaum Baum))
                    (AVL-Baum?
                      (rechter-Teilbaum Baum))
                    (= (Baum-Hoehe Baum)
                       (+ 1 (max
                              (Baum-Hoehe
                                (linker-Teilbaum Baum))
                              (Baum-Hoehe
                                (rechter-Teilbaum Baum)))))
                    (<= (Balance Baum) 1)
                    (>= (Balance Baum) -1)
                    (or (leerer-Baum?
                          (linker-Teilbaum Baum))
                        (string-ci<?
                           (Baum-Key
                             (linker-Teilbaum Baum))
                           (Baum-Key Baum)))
                    (or (leerer-Baum?
```

```
                    (rechter-Teilbaum Baum))
                  (string-ci<?
                     (Baum-Key Baum)
                     (Baum-Key
                       (rechter-Teilbaum Baum)))))
          Baum)
         (#t #f))))
 ;;Zur Bestimmung der Ausgeglichenheit
(Balance
 (lambda (Baum)
   (cond((leerer-Baum? Baum) 0)
         (#t (- (Baum-Hoehe
                  (rechter-Teilbaum Baum))
                (Baum-Hoehe
                   (linker-Teilbaum Baum)))))))

(*leerer-Baum* (Make-Baum 0 "" "" null null))

(leerer-Baum?
 (lambda (Baum)
   (and (vector? Baum)
        (= (vector-length Baum) 5)
        (= (Baum-Hoehe Baum) 0)
        (= (string-length
             (Baum-Key Baum)) 0)
        (= (string-length
             (Baum-Text Baum)) 0)
        (null? (linker-Teilbaum Baum))
        (null? (rechter-Teilbaum Baum)))))

(Baum-Hoehe (lambda (Baum)
  (vector-ref Baum 0)))
(Baum-Key (lambda (Baum)
  (vector-ref Baum 1)))
(Baum-Text (lambda (Baum)
 (vector-ref Baum 2)))

(linker-Teilbaum
 (lambda (Baum)
   (vector-ref Baum 3)))

(rechter-Teilbaum
 (lambda (Baum)
   (vector-ref Baum 4)))

(get-key-AVL-Baum
 (lambda (key Baum)
   (cond((leerer-Baum? Baum) Baum)
         ((string-ci=? key (Baum-Key Baum))
            Baum)
         ((string-ci<? key (Baum-Key Baum))
          (get-key-AVL-Baum key
            (linker-Teilbaum Baum)))
         ((string-ci<? (Baum-Key Baum) key)
          (get-key-AVL-Baum key
            (rechter-Teilbaum Baum)))
```

```
         (#t (display
               "Fehler bei get-key-AVL-Baum!")
             (display Baum)
             #f))))

(get-minimum-key-AVL-Baum
 (lambda (Baum)
   (cond((leerer-Baum? Baum) Baum)
         ((leerer-Baum? (linker-Teilbaum Baum))
            Baum)
         (#t (get-minimum-key-AVL-Baum
              (linker-Teilbaum Baum))))))

(get-maximum-key-AVL-Baum
 (lambda (Baum)
   (cond((leerer-Baum? Baum) Baum)
         ((leerer-Baum? (rechter-Teilbaum Baum))
            Baum)
         (#t (get-maximum-key-AVL-Baum
               (rechter-Teilbaum Baum))))))

(get-next-key-AVL-Baum
 (lambda (key Baum)
   (cond((leerer-Baum? Baum) Baum)
         ((string-ci<? key (Baum-Key Baum))
          (let ((l-teil
                  (get-next-key-AVL-Baum
                     key
                     (linker-Teilbaum Baum))))
            (cond((leerer-Baum? l-teil) Baum)
                 (#t l-teil))))
         ((string-ci=? key (Baum-Key Baum))
          (get-minimum-key-AVL-Baum
             (rechter-Teilbaum Baum)))
         ((string-ci<? (Baum-Key Baum) key)
          (get-next-key-AVL-Baum
             key
             (rechter-Teilbaum Baum)))
         (#t (display
           "Fehler bei get-next-key-AVL-Baum: ")
             (display Baum)
             #f))))

(AVL-Baum-montage
 (lambda (Baum l_Teilbaum r_Teilbaum)
   (let* ((l-hoehe (Baum-Hoehe l_Teilbaum))
          (r-hoehe (Baum-Hoehe r_Teilbaum))
          (knotenbildung
           (lambda (key text l_Baum r_Baum)
             (Make-Baum
                (+ 1 (max
                       (Baum-Hoehe l_Baum)
                       (Baum-Hoehe r_Baum)))
                key text l_Baum r_Baum))))
     (cond(;;ausgeglichen
            (and (< l-hoehe (+ r-hoehe 2))
```

```
      (< r-hoehe (+ l-hoehe 2)))
 (knotenbildung
    (Baum-Key Baum)
    (Baum-Text Baum)
    l_Teilbaum
    r_Teilbaum))

 ;;Fall 1
((and (> l-hoehe r-hoehe)
      (< (Balance l_Teilbaum) 1))
 (let* ((l-l-teil
          (linker-Teilbaum
            l_Teilbaum))
        (r-l-teil
          (rechter-Teilbaum
            l_Teilbaum))
        (r-l-teil&r-teil
         (knotenbildung
            (Baum-Key Baum)
            (Baum-Text Baum)
            r-l-teil
            r_Teilbaum)))
   (knotenbildung
      (Baum-Key l_Teilbaum)
      (Baum-Text l_Teilbaum)
      l-l-teil
      r-l-teil&r-teil)))

 ;;Fall 2
((and (> l-hoehe r-hoehe)
      (= (Balance l_Teilbaum) 1))
 (let* ((neue-wurzel
          (rechter-Teilbaum
            l_Teilbaum))
        (l-r-l-teil
           (linker-Teilbaum
             neue-wurzel))
        (l-l-teil
           (linker-Teilbaum
             l_Teilbaum))
        (r-r-l-teil
           (rechter-Teilbaum
             neue-wurzel))
        (l-l-teil&l-r-l-teil
         (knotenbildung
            (Baum-Key l_Teilbaum)
            (Baum-Text l_Teilbaum)
            l-l-teil l-r-l-teil))

        (r-r-l-teil&r-teil
         (knotenbildung
            (Baum-Key Baum)
            (Baum-Text Baum)
            r-r-l-teil r_Teilbaum)))
   (knotenbildung
      (Baum-Key neue-wurzel)
```

```
                    (Baum-Text neue-wurzel)
                    l-l-teil&l-r-l-teil
                     r-r-l-teil&r-teil)))

                ;;Fall 3
                ((and (< l-hoehe r-hoehe)
                      (> (Balance r_Teilbaum) -1))
                 (let* ((r-r-teil
                           (rechter-Teilbaum r_Teilbaum))
                        (l-r-teil
                          (linker-Teilbaum r_Teilbaum))
                        (l-teil&l-r-teil
                         (knotenbildung
                            (Baum-Key Baum)
                            (Baum-Text Baum)
                            l_Teilbaum l-r-teil)))
                   (knotenbildung
                      (Baum-Key r_Teilbaum)
                      (Baum-Text r_Teilbaum)
                      l-teil&l-r-teil r-r-teil)))

                ;;Fall 4
                ((and (< l-hoehe r-hoehe)
                      (= (Balance r_Teilbaum) -1))
                 (let* ((neue-wurzel
                           (linker-Teilbaum r_Teilbaum))
                        (r-l-r-teil
                          (rechter-Teilbaum
                            neue-wurzel))
                        (r-r-teil
                          (rechter-Teilbaum
                            r_Teilbaum))
                        (l-l-r-teil
                          (linker-Teilbaum
                            neue-wurzel))
                        (r-l-r-teil&r-r-teil
                         (knotenbildung
                            (Baum-Key r_Teilbaum)
                            (Baum-Text r_Teilbaum)
                            r-l-r-teil r-r-teil))
                        (l-teil&l-l-r-teil
                         (knotenbildung
                            (Baum-Key Baum)
                            (Baum-Text Baum)
                            l_Teilbaum l-l-r-teil)))
                   (knotenbildung
                      (Baum-Key neue-wurzel)
                      (Baum-Text neue-wurzel)
                      l-teil&l-l-r-teil
                      r-l-r-teil&r-r-teil)))))))

(einfuegen-in-AVL-Baum
 (lambda (key text Baum)
   (cond((leerer-Baum? Baum)
         (Make-Baum
            1 key text
```

```
                  *leerer-Baum* *leerer-Baum*))
               ((string-ci=? key (Baum-Key Baum))
                (Make-Baum
                   (Baum-Hoehe Baum)
                   (Baum-Key Baum)
                   text
                   (linker-Teilbaum Baum)
                   (rechter-Teilbaum Baum)))
               ((string-ci<? key (Baum-Key Baum))
                (AVL-Baum-montage
                   Baum
                   (einfuegen-in-AVL-Baum
                      key text
                      (linker-Teilbaum Baum))
                   (rechter-Teilbaum Baum)))
               ((string-ci<? (Baum-Key Baum) key)
                (AVL-Baum-montage
                   Baum
                   (linker-Teilbaum Baum)
                   (einfuegen-in-AVL-Baum
                      key text
                      (rechter-Teilbaum Baum)))))))

      (AVL-Baum-struktur-zeigen
       (lambda (Baum anzahl_space)
         (letrec
            ((write-space
              (lambda (n)
                (cond((= n 0) null)
                      (#t (write-string " ")
                         (write-space
                           (- n 1))))))
             (zeigen
              (lambda (Baum eingerueckt)
                (cond((leerer-Baum? Baum)
                                    (void))
                      (#t (zeigen
                            (rechter-Teilbaum Baum)
                            (+ eingerueckt
                               anzahl_space))
                         (write-space eingerueckt)
                         (display
                            (Baum-Key Baum))
                         (display " /")
                         (display
                            (Baum-Hoehe Baum))
                         (display "/ ")
                         (newline)
                         (zeigen
                            (linker-Teilbaum Baum)
                            (+ eingerueckt
                               anzahl_space)))))))
           (zeigen Baum anzahl_space)))))

;;;;Auswahl des Benutzerkommandos
;;;; (dispatch function)
```

```
(cond((eq? operation 'Start)
       (set! *DATEN*
             (with-input-from-file
                (car parameter)
                (lambda ()
                  (read))))
       #t)
      ((eq? operation 'Ende)
       (with-output-to-file
          (car parameter)
          (lambda ()
            (write *DATEN*)))
       #t)
      ((AVL-Baum? *DATEN*)
       (cond((eq? operation 'Eingabe)
             (set! *DATEN*
                   (einfuegen-in-AVL-Baum
                      (car parameter)
                      (cadr parameter)
                      *DATEN*))
             #t)
            ((eq? operation 'Get-Text)
             (Baum-Text
                (get-key-AVL-Baum
                  (car parameter) *DATEN*)))
            ((eq? operation 'Get-min-Key)
             (Baum-Key
                (get-minimum-key-AVL-Baum *DATEN*)))
            ((eq? operation 'Get-max-Key)
             (Baum-Key
               (get-maximum-key-AVL-Baum *DATEN*)))
            ((eq? operation 'Get-next-Key)
             (if (string-ci=?
                    (car parameter)
                    (Baum-Key
                      (get-maximum-key-AVL-Baum
                      *DATEN*)))
                 (begin (display
               "Hinweis: Eingabe war groesster key!")
                    (car parameter))
                 (Baum-Key (get-next-key-AVL-Baum
                      (car parameter) *DATEN*))))
            ((eq? operation 'Struktur-zeigen)
             (AVL-Baum-struktur-zeigen *DATEN* 2))
            (#t (display
                  "Operation nicht implementiert: ")
                (display operation))))
      ((and (null? *DATEN*)
            (eq? operation 'Eingabe))
       (set! *DATEN*
             (einfuegen-in-AVL-Baum
                (car parameter) (cadr parameter)
                *leerer-Baum*))  #t)
      (#t (display "Keine wohlstrukturierten Daten : ")
          (display *DATEN*)
          )))))
```

Beispiel einer Applikation des Programms `Aktenverwaltung` (↪ Codebeginn S. 229):

```
eval> (define *DATEN* (list))
eval> (begin
       (Aktenverwaltung 'Eingabe
         "L-a-2" "Fall Meyer")
       (Aktenverwaltung 'Eingabe
         "L-a-2/b" "Fortsetzung Meyer")
       (Aktenverwaltung 'Eingabe
         "G-b-3-2" "Plan Westfeld")
       (Aktenverwaltung 'Eingabe
         "Y-345-6" "Fall Schulze")
       (Aktenverwaltung 'Eingabe
         "Dortmund" "Stadtplanung")
       (Aktenverwaltung 'Eingabe
         "Herbert" "Fall Herbert")
       (Aktenverwaltung 'Eingabe
         "A-1-122" "Rechnerbeschaffung")
       (Aktenverwaltung 'Eingabe
         "G-b-3-2" "Plan Neu-Westfeld")
       (Aktenverwaltung 'Ende
         "D:\\bonin\\scheme\\AVL.DAT"))
      ==> #t
eval> (Aktenverwaltung 'Struktur-zeigen)
      Y-345-6 /1/
    L-a-2/b /2/
  L-a-2 /4/
      Herbert /1/
    G-b-3-2 /3/
      Dortmund /2/
        A-1-122 /1/
```

2.3.4 Zusammenfassung: Vektor

Ein Vektor ist eine Struktur mit konstanter Länge. Seine Elemente können beliebige symbolische Ausdrücke sein, also auch wieder Vektoren. Ein Vektor hat sich selbst als Wert (EVAL-Regel 1, ↪ Abschnitt 1.1.2 S. 21). Die Zugriffszeit ist unabhängig davon, ob auf ein Element mit kleinem oder großem Index zugegriffen wird. Für einen dünn besetzten Vektor ist die A-Liste eine zweckmäßige Alternative (↪ Abschnitt 2.2.2 S. 183).

Charakteristische Beispiele für Abschnitt 2.3.1

```
eval> ;;;;Verwaltung von Adressen
```

```
      (define Adressen-Struktur
        #("Anrede & Titel"
          "Vorname & Name"
          "Strasse & Nummer"
          "Postfach"
          "Landeskenner & Postleitzahl & Ortsname"))

eval> (define Privat-Adresse
        (lambda (Anrede Name Strasse Ort)
          (let ((Adresse
                 (make-vector
                    (vector-length
                      Adressen-Struktur)
                    #f)))
            (vector-set! Adresse 0 Anrede)
            (vector-set! Adresse 1 Name)
            (vector-set! Adresse 2 Strasse)
            (vector-set! Adresse 4 Ort)
            Adresse)))
eval> (define Krause
       (Privat-Adresse
           "Herr"
           "Hans Krause"
           "Entenweg 7"
           "D-21391 Reppenstedt"))

eval> (define Druck-Adresse
        (lambda (Adresse)
          (let ((V-Laenge
                 (vector-length Adresse)))
            (do ((i 0 (+ i 1)))
                ((= i V-Laenge) (void))
              (cond ((vector-ref Adresse i)
                       (begin
                         (display
                           (vector-ref
                             Adresse i))
                           (newline)))
                    (#t ))))))

eval> Krause ==>
       #("Herr" "Hans Krause" "Entenweg 7"
         #f "D-21391 Reppenstedt")

eval> (Druck-Adresse Krause)  ==>
      Herr
      Hans Krause
      Entenweg 7
      D-21391 Reppenstedt
```

2.4 Abbildungsoption: Zeichenkette und Symbol

Eine Zeichenkette (engl.: *string*) ist eine Sequenz aus einzelnen Zeichen (Elementen). Im Unterschied zur Liste oder zum Vektor können die Elemente dieser Sequenz jedoch nicht wieder Zeichenketten sein. Eine Zeichenkette hat sich selbst als Wert (*EVAL*-Regel 1, ↪ Abschnitt 1.1.2 S. 21). Bisher haben wir daher eine Zeichenkette als Konstante benutzt.

In diesem Abschnitt konstruieren, selektieren und manipulieren wir solche „Konstanten“. Die Zeichenkette ist damit eine eigenständige Abbildungsoption wie die Liste oder der Vektor. Als Beispiel dient die Aufgabe zwei Muster miteinander zu vergleichen (engl.: *pattern matching*). Ein entsprechendes Konstrukt realisieren wir zunächst mit der Abbildungsoption Liste (↪ Abschnitt 2.4.1 S. 239). Dieses `Simple-Match`-Konstrukt arbeitet „naturgemäß“ mit `car`- und `cdr`-Selektoren. Um Zeichenketten vergleichen zu können, implementieren wir entsprechende `car-String`- und `cdr-String`-Konstrukte mit Hilfe der eingebauten `string`-Konstrukte (↪ Abschnitt 2.4.2 S. 241). LISP-Systeme verfügen über ein große Zahl spezieller `string`-Konstrukte. Wir wollen hier primär den Aspekt Abbildungsoption vertiefen und erwähnen daher nur einige Konstrukte aus dem `string`-Konstrukte-Repertoire.

Im Rahmen des Vergleichs zweier Muster führen wir Variable (engl.: *match variables*) ein. Das so erweiterte Konstrukt `Mustervergleich` nutzt die Repräsentation des Symbols, d. h. das Ergebnis der *PRINT*-Phase (kurz: *PRINT*-Name) als Informationsträger. Bisher haben wir ein Symbol mit einem Wert oder einer Eigenschaft (P-Liste) verknüpft. Der im *READ-EVAL-PRINT*-Zyklus (↪ Abschnitt 1.1.2 S. 20) zurückgegebene *PRINT*-Name diente zur Identifizierung. Er wurde als eine Konstante (Literalatom) betrachtet. Hier wird dieser *PRINT*-Name selbst selektiert und neu konstruiert (↪ Abschnitt 2.4.3 S. 248).

Dass leistungsfähige `match`-Konstrukte einen eigenständigen „Denkrahmen“ (↪ Abschnitt 2.1 S. 141) bilden, skizzieren wir mit einem kleinen Vorübersetzer („Precompiler“ oder „Präprozessor“). Er setzt Pseudocode-Formulierungen in LISP-Code um. Er transformiert z. B. eine `while` ... `do` ... `od`-Iteration (↪ S. 89) in ein rekursives LISP-Konstrukt. Dazu ist ein neues Symbol zu generieren, das keine Namensgleichheit mit anderen Symbolen hat. Es darf nicht mit schon existierenden Symbolen bzw. mit Symbolen, die als spätere Eingabe vorkommen, verwechselbar sein. Ein solches, einmaliges Symbol generiert das eingebaute Konstrukt `gensym` (↪ Abschnitt 2.4.4 S. 259).

2.4.1 Mustervergleich

Programme, die Muster miteinander vergleichen, sind ein klassisches Anwendungsgebiet für LISP. Die LISP-Literatur befasst sich eingehend mit dem Thema *Pattern Matching* (↪ z. B. [135] S. 151 ff; [178] S. 235 ff oder [190] S. 253 ff).

Dabei ist anzumerken, dass der Terminus „Mustervergleich“ eine unzureichende Übersetzung darstellt. Er vermittelt spontan den Eindruck, es wird nur verglichen im Sinne einer Klassifikation. Vielmehr zielt *Pattern Matching* jedoch auf ein Paradigma, auf einen eigenständigen „Denkrahmen“ für die Problemanalyse und Lösungsfindung.

Ein Muster beschreibt den Typ einer „Einheit“. Beim Vergleich wird entschieden, ob ein symbolischer Ausdruck „passt“, d. h. ob er dieser Typbeschreibung entspricht. Ein Muster selbst besteht aus einer linearen Folge von Elementen. Das Muster wird von links nach rechts mit dem zu prüfenden symbolischen Ausdruck verglichen. Wir nehmen zunächst an, dass das Muster und der zu prüfende symbolische Ausdruck Listen sind.

Die Listenelemente unseres Musters können sein:

- ? (ein Fragezeichen),
- * (ein Stern) oder
- ein zu erfüllendes Kennwort (ein sonstiges Symbol).

Ein Fragezeichen im Muster steht für ein beliebiges Element im Prüfling. Anders formuliert: Das Fragezeichen deckt ein Element. Der Stern deckt eine Folge von beliebigen Elementen (incl. einer leeren Folge). Er deckt ein ganzes Segment. Jedes andere Symbol im Muster deckt nur das gleiche Symbol im Prüfling. Ein solch einfacher Mustervergleich kann z. B. folgende Entscheidungen treffen:

```
eval> (Simple-Match
        '(Eine ? Analyse sichert *)      ;Muster
        '(Eine zweckmaessige Analyse     ;Prüfling
          sichert den Automationserfolg)
       ) ==> #t                          ;Ergebnis


eval> (Simple-Match
        '(Eine * Analyse sichert ?)      ;Muster
        '(Eine zweckmaessige Analyse     ;Prüfling
          sichert den Automationserfolg)
        ) ==> #f                         ;Ergebnis
```

Für den Vergleich von Listen definieren wir das `Simple-Match`-Konstrukt als eine rekursive Funktion, wie folgt:

```
eval> (define Simple-Match (lambda (Muster Liste)
        (cond((and (null? Muster)
```

```
      (null? Liste)) #t)
((and (null? Liste)
      (null? (cdr Muster))
      (eq? '* (car Muster)))
                     #t)
 ((or (null? Muster)
      (null? Liste)) #f)
 ((or (eq? (car Muster)
           (car Liste))
      (eq? '? (car Muster)))
      (Simple-Match
        (cdr Muster) (cdr Liste)))
 ((eq? '* (car Muster))
  (cond((Simple-Match
          (cdr Muster)
          (cdr Liste)) #t) ;*-Fall a
       ((Simple-Match
          Muster
         (cdr Liste)) #t) ;*-Fall b
       ((Simple-Match
          (cdr Muster)
          Liste) #t) ;*-Fall c
       (#t #f)))     ;*-Fall d
 (#t #f))))
```

Die erste Klausel des ersten obigen `cond`-Konstruktes prüft, ob das Muster (hier `lambda`-Variable `Muster`) und der Prüfling (hier `lambda`-Variable `Liste`) zum gleichen Zeitpunkt abgearbeitet sind; also faktisch gleich lang sind. Ist die Liste abgearbeitet und das letzte Element von Muster ein Stern, dann passt das Muster, da wir unterstellen, dass der Stern auch ein leeres Segment deckt. Ist die Liste kürzer, dann kann das Muster nicht passen (3. Klausel). Ist das erste Element im Muster gleich dem ersten Element im Prüfling, dann entscheidet der Mustervergleich der Restlisten, ob das Muster passt oder nicht (4. Klausel). Ist das erste Element im Muster das Fragezeichen, dann passt es auf jedes erste Element im Prüfling. Auch dann entscheidet der Mustervergleich der Restlisten (daher auch 4. Klausel). Ist das erste Element des Musters ein Stern, dann sind folgende Fälle zu unterscheiden (5. Klausel):

1. Der Stern steht für ein Element im Prüfling (`*-Fall a`). Der Mustervergleich ist mit den Restlisten fortzusetzen. Ist der Vergleich der Restlisten erfolgreich, dann passt das Muster.

2. Der Stern steht für mehrere Elemente im Prüfling (`*-Fall b`). Wir behalten das aktuelle Muster bei, d. h. verkürzen es nicht und vergleichen es mit dem Rest des Prüflings. Dadurch ist es beim erneuten Vergleich (`car Muster`) wieder der Stern, der dann mit dem nächsten Element des Prüflings verglichen wird.

3. Der Stern steht für kein Element im Prüfling (`*-Fall c`). Wir verkürzen das Muster, so dass wir den Stern „aufgeben“, und vergleichen das verkürzte Muster mit dem aktuellen Prüfling.

4. Die rekursiven `Simple-Match`-Anwendungen stehen im `cond`-Konstrukt als Prädikate. Führt keines zu einem Wert ungleich `#f`, dann passt das Muster nicht. Zur leichteren Erkennbarkeit ist eine entsprechende `#t`-Klausel formuliert (`*-Fall d`).

2.4.2 `string`-Konstrukte

Beispiel: Einfacher Mustervergleich für Zeichenketten

Im bisherigen `Simple-Match`-Konstrukt (↪ Abschnitt 2.4.1 S. 239) sind das Muster und der Prüfling Listen. Wir entwickeln daraus ein `Simple-Match`-Konstrukt, das mit Zeichenketten arbeitet. Statt Symbole in einer Liste zu prüfen, sind Teile von Zeichenketten zu prüfen. Das obige Beispiel ist für dieses string-basierte `Simple-Match`, wie folgt, zu notieren:

```
eval> (Simple-Match
 "Eine ? Analyse sichert *"
 "Eine zweckmaessige Analyse sichert
                         den Automationserfolg"))
       ==> #t
```

An die Stelle des Prädikates `null?` für die Liste tritt das Prädikat `string-null?`, das wir auf der Basis der Konstrukte `string-length` und `string?` wie folgt definieren:

```
eval> (define string-null?
        (lambda (String)
          (and
           (string? String)
           (= (string-length String) 0))))
eval> (string-null? "") ==> #t
eval> (string-null? 7) ==> #f
```

Das Prädikat `eq?` wird durch das eingebaute Prädikat `string=?` ersetzt. Der Wert des `string-null?`-Konstruktes ist nur dann ungleich `false`, also `true`, wenn die Länge der Zeichenkette gleich Null ist. Das `string=?`-Konstrukt vergleicht seine beiden Argumente Zeichen für Zeichen. Es hat einen Wert `true`, wenn für jedes Zeichen Gleichheit gilt und die beiden Zeichenketten gleich lang sind. Die folgenden Beispiele zeigen die beiden Prädikate für Zeichenketten.

```
eval> (string #\a) ==> "a"
eval> (string) ==> ""
eval> (string-null? (string)) ==> #t
eval> (define Foo " ")
```

```
eval> (string-length Foo) ==> 1
eval> (string-null? Foo) ==> #f
eval> (string=? "more" "more") ==> #t
eval> (string=? "MoRe" "more") ==> #f
eval> (string=? "MoRe " "MoRe") ;Länge
      ==> #f                    ; ungleich
```

Zum Ersatz der Selektoren `car` und `cdr` sind entsprechende Selektoren für eine Zeichenkette selbst zu definieren. Wir nennen sie `car-string` und `cdr-string`. Dazu betrachten wir das Zeichen Zwischenraum, allgemein auch *Space* oder *Blank* genannt, als Trennzeichen für die Zeichenteilketten. Der Zwischenraum als einzelnes Zeichen (engl.: *character*) hat die Repräsentation `#\space`. Ob eine Zeichenkette vorliegt, stellt das Prädikat `string?` fest; ob ein einzelnes Zeichen vorliegt, das Prädikat `char?`. Ein Zeichen einer Zeichenkette selektiert das `string-ref`-Konstrukt. Es ist, wie folgt, definiert:

`eval> (string-ref` < *string* >< *position* >`)`
`==>` < *character* >

mit:

< *string* >	≡ Eine Zeichenkette.
< *position* >	≡ Eine ganzzahlige, nicht negative Angabe der Position des zu selektierenden Zeichens. Die Zählung der Position beginnt bei 0. Bedingung ist: `(and (<= 0` < *position* >`)` `(<` < *position* > `(string-length` < *string* >`)))`
< *character* >	≡ Das selektierte Zeichen der Zeichenkette. Achtung! Es ist vom Typ Zeichen (engl.: *character*) nicht vom Typ Zeichenkette (engl.: *string*); ↪ dazu die folgenden Beispiele.

```
eval> (define Foo
        "Requirements Engineering")
eval> (string-length Foo) ==> 24
eval> (string-ref Foo 3) ==> #\u
eval> (string-ref Foo 24) ==> ERROR ...
      ;string-ref: index 24 out of range [0, 23]
eval> (string-ref Foo 23) ==> #\g
eval> (string-ref Foo 12) ==> #\space
eval> (string? Foo) ==> #t
eval> (string? (string-ref Foo 12)) ==> #f
eval> (char? (string-ref Foo 12)) ==> #t
eval> (char? Foo) ==> #f
```

Der Wert des `string-ref`-Konstruktes ist vom Typ Zeichen, d.h. das Prädikat `char?` ist erfüllt, und nicht vom Typ Zeichenkette, d. h.

das Prädikat `string?` wird nicht erfüllt. Zum Prüfen eines mit dem `string-ref`-Konstrukt selektiertem Zeichen auf `#\space`, können wir daher nicht das Prädikat `string=?` nutzen. Es ist mit dem Prädikat `char=?` zu prüfen. Wir definieren daher ein Konstrukt `space?`, das prüft, ob an einer bestimmten Stelle ein Zwischenraum in der Zeichenkette gegeben ist, wie folgt:

```
eval> (define space?
        (lambda (String Position)
          (char=?
           (string-ref String Position)
           #\space)))
```

Analog zu einer Liste, bei der zur Trennung der Elemente ein Zwischenraum ausreicht, jedoch beliebig viele Zwischenräume stehen können, fassen wir für das Selektieren einer Zeichenteilkette ebenfalls beliebig viele Zwischenräume als ein Trennzeichen auf. Für die zu definierenden Konstrukte `car-string` und `cdr-string` unterstellen wir folgende Ergebnisse:

```
eval> (car-string "   more LISP ")
        ==> "more"    ;Keine führenden
                      ; und nachfolgenden
                      ; Zwischenräume.
eval> (cdr-string "   more LISP ")
        ==> " LISP "  ;Führende und
                      ; nachfolgende
                      ; Zwischenräume
                      ; bleiben erhalten.
```

Die Annahme, dass mehrere Zwischenräume als ein Trennzeichen gewertet werden, erleichtert das Formulieren des Musters. Ein Nachteil ist jedoch, dass wir die Zusicherung aufgeben, aus den selektierten Teilen wieder das Ganze in ursprünglicher Form erstellen zu können (↪ Abschnitt 2.1.2 S. 146). Das `string-append`-Konstrukt verbindet zwei Zeichenketten zu einer. Wenden wir dieses Konstrukt auf die selektierten Zeichenteilketten an, dann fehlen dem Ergebnis die Zwischenräume, die zu Beginn der Zeichenkette standen.

```
eval> (string-append
        (car-string "   more LISP ")
        (cdr-string "   more LISP "))
      ==> "more LISP "
```

Unsere Selektoren `car-string` und `cdr-string` nutzen den eingebauten Selektor `substring`. Das `substring`-Konstrukt ist, wie folgt, definiert:

`eval>` (`substring` < *string* >< *start* >< *ende* >)
`==>` < *substring* >

mit:

< *string* > ≡ Eine Zeichenkette.

< *start* > ≡ Eine ganzzahlige, nicht negative Angabe der Position des ersten Zeichens der zu selektierenden Zeichenkette. Die Zählung der Position beginnt bei 0. Bedingung ist:

```
(and (<= 0 < position >)
     (<= < start >
      (string-length < string >)))
```

< *ende* > ≡ Eine ganzzahlige, nicht negative Angabe der Position des ersten Zeichens, das nicht mehr zur Zeichenkette gehören soll. Die Zählung der Position beginnt bei 0. Bedingung ist:

```
(and (<= 0 < ende >)
     (<= < ende >
      (string-length < string >))
     (<= < start >< ende >))
```

< *substring* > ≡ Die selektierte Zeichenteilkette. Sie beginnt mit dem Zeichen bei < *start* > und endet mit dem Zeichen bei `(- < ende > 1 )`; ↪ dazu die folgenden Beispiele.

```
eval> Foo ==>
      "Requirements Engineering"
eval> (substring Foo 0 12) ==> "Requirements"
eval> (substring Foo 0 24) ==>
      "Requirements Engineering"
eval> (substring Foo 24 24) ==> ""
eval> (string-null? (substring Foo 13 13)) ==> #t
eval> (substring Foo 13 12) ==> ERROR ...
      ;substring: ending index 12
      ; out of range [13, 24] ...
eval> (substring Foo 0 27) ==> ERROR ...
      ;substring: ending index 27
      ; out of range [0, 24] ...
```

Das `car-string`-Konstrukt definieren wir als eine LISP-Funktion mit dem eigentlichen Selektionskonstrukt `Von-bis-Analyse` und den Hilfsgrößen wie z. B. `space?` als lokale Funktionen. Die rekursive `Von-bis-Analyse` ermittelt für die Anwendung des `substring`-Konstruktes die < *start* >- und < *ende* >-Werte. Sie arbeitet mit zwei Zeigern. Der erste Zeiger ist der Wert ihrer `lambda`-Variablen `Car-Start`. Dieser Zeiger wird auf das erste Zeichen, das ungleich `#\space` ist, gesetzt. Der zweite Zeiger ist der Wert ihrer `lambda`-Variablen `Aktuelle-Position`. Dieser wird so lange verschoben, bis er auf dem ersten Zwischenraum nach einem Zeichen ungleich `#\space` steht oder das Ende der Zeichenkette erreicht wurde.

Zum Überspringen von führenden Zwischenräumen verwenden wir die lokale Variable `Vorspann-Space?` im Sinne eines Schalters. Zunächst ist ihr Wert `#t`. Wird das erste Zeichen ungleich `#\space` erkannt, erhält sie den Wert `#f`. Ist ihr Wert `#f` und wird ein Zeichen gleich `#\space` erkannt, dann ist das Ende der Zeichenteilkette erreicht. Zu beachten ist, dass das `substring`-Konstrukt als < *ende* >-Wert die Position nach unserer `car-string`-Zeichenteilkette benötigt. Wir definieren somit folgendes `car-string`-Konstrukt:

```
eval> (define car-string (lambda (String)
        (letrec
          ((Ende-Position
             (- (string-length String) 1))
           (Vorspann-Space? #t)
           (space? (lambda (Pos)
             (char=? (string-ref String Pos)
                     #\space)))
           (Von-bis-Analyse
            (lambda (Car-Start Aktuelle-Position)
             (cond((<= Aktuelle-Position
                       Ende-Position)
                   (cond((and Vorspann-Space?
                              (space?
                                Aktuelle-Position))
                         (Von-bis-Analyse
                           (+ Car-Start 1)
                           (+ Aktuelle-Position
                             1)))
                        ((space? Aktuelle-Position)
                           (substring
                             String
                             Car-Start
                             Aktuelle-Position))
                        (#t
                           (set! Vorspann-Space?
                                 #f)
                           (Von-bis-Analyse
                             Car-Start
                             (+ Aktuelle-Position
                                1)))))
                  ((= Aktuelle-Position
                      (+ Ende-Position 1))
                    (substring
                      String
                      Car-Start
                      Aktuelle-Position))
                   (#t
                    (display
                      "Kein car-string bestimmt: ")
```

```
                    (display String))))))
          (Von-bis-Analyse 0 0))))

eval> (car-string "  Alles klar?") ==> "Alles"
```

Das `cdr-string`-Konstrukt ist analog zum `car-string`-Konstrukt aufgebaut. Die eigentliche Berechnung der Positionen für das `sub-string`-Konstrukt vollzieht die lokale Funktion `Von-Analyse`. Hier ist nur die Anfangsposition der zu selektierenden Zeichenteilkette durch den Zeiger `Aktuelle-Position` zu ermitteln. Die Endeposition ist identisch mit der gesamten Zeichenkette; sie ist mit dem `string-length`-Konstrukt bestimmbar.

Wenn eine Zeichenkette leer ist oder nur eine Zeichenteilkette für das `car-string`-Konstrukt enthält, ist der Wert des `cdr-string`-Konstruktes die leere Zeichenkette. Im Sinne unseres Verbundes aus Konstruktor, Selektor, Prädikat und Mutator erstellen wir die leere Zeichenkette mit dem Konstruktor `make-string`. Das `make-string`-Konstrukt ist, wie folgt, definiert:

`eval>` (`make-string` < *length* > {< *one* − *character* >})
`==>` < *string* >

mit:

< *length* >	≡ Eine ganzzahlige, nicht negative Angabe der Länge der Zeichenkette. Es gibt eine Obergrenze für die Länge. Z. B. besteht in *PC Scheme* die Restriktion: `(and (<= 0` < *length* > `) (<=` < *length* > `16380))`
< *one* − *character* >	≡ Ein einzelnes Zeichen als Füllzeichen für die Zeichenkette. Ist kein Füllzeichen angegeben wird `#\space` verwendet.
< *string* >	≡ Eine Zeichenkette mit der angegebenen Länge.

```
eval> (make-string 5 #\g) ==> "ggggg"
eval> (string-null? (make-string 0)) ==> #t
eval> (make-string (* 2 3) #\a) ==> "aaaaaa"
eval> (define strich #\|)
eval> (make-string 4 strich) ==> "||||"
```

Wir definieren damit das `cdr-string`-Konstrukt, wie folgt:

```
eval> (define cdr-string (lambda (String)
        (letrec
          ((Ende-Position
             (- (string-length String) 1))
           (Vorspann-Space? #t)
           (space?
            (lambda (Pos)
```

```
              (char=? (string-ref String Pos)
                      #\space)))
            (Von-Analyse
             (lambda (Aktuelle-Position)
              (cond((<= Aktuelle-Position
                        Ende-Position)
                    (cond((and
                           Vorspann-Space?
                           (space? Aktuelle-Position))
                          (Von-Analyse
                            (+ Aktuelle-Position 1)))
                         ((space? Aktuelle-Position)
                            (substring
                              String
                              Aktuelle-Position
                              (+ Ende-Position 1)))
                         (#t (set!
                               Vorspann-Space? #f)
                            (Von-Analyse
                              (+ Aktuelle-Position
                                 1)))))
                   ((= Aktuelle-Position
                       (+ Ende-Position 1))
                    (make-string 0))
                   (#t (display
                         "Kein cdr-string bestimmt: ")
                    (display string))))))
           (Von-Analyse 0))))
;
eval> (cdr-string "  Alles klar?") ==> " klar?"
```

Mit den erstellten Selektoren `car-string` und `cdr-string` können wir den obigen Mustervergleich für Zeichenketten formulieren. Das `Simple--Match`-Konstrukt hat dann folgende Definition:

Programm: `Simple-Match`

```
eval> (define Simple-Match (lambda (Muster  String)
     (cond((and
            (string-null? Muster)
            (string-null? String)) #t)
          ((and (string-null? String)
                (string-null? (cdr-string Muster))
                (string=? "*" (car-string Muster))) #t)
          ((or (string-null? Muster)
               (string-null? String)) #f)
          ((or (string=? (car-string Muster)
                         (car-string String))
               (string=? "?" (car-string Muster)))
                 (Simple-Match (cdr-string Muster)
                               (cdr-string String)))
```

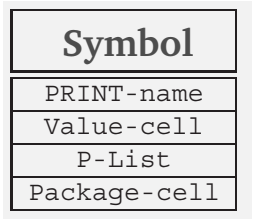
```
          ((string=? "*" (car-string Muster))
            (cond((Simple-Match (cdr-string Muster)
                                (cdr-string String)) #t)
                 ((Simple-Match Muster
                                (cdr-string String)) #t)
                 ((Simple-Match (cdr-string Muster)
                                string) #t)
                 (#t #f)))
     (#t #f))))

eval> (Simple-Match
        "Eine ? Analyse sichert *"
        "Eine Analyse sichert Erfolg")
       ==> #f   ;Kein Element für das ?.
eval> (Simple-Match
        "* haben * Wert"
        "Viele Symbole haben einen komplexen Wert")
      ==> #t
eval> (Simple-Match
        "? Phasenplan bestimmt * Vorgehen"
        "Der lineare Phasenplan bestimmt das Vorgehen")
        ==> #f  ;Mehr als ein Element für das ?.
eval> (Simple-Match
        "* Phasenplan bestimmt * Vorgehen"
        "Der lineare Phasenplan bestimmt das Vorgehen")
      ==> #t
```

2.4.3 Symbol und *PRINT*-Name

Ein Symbol (Literalatom) ist eine „Einheit“ mit einer eigenen Identität (↪ S. 42). Seine Elementarteilchen sind abhängig vom jeweiligen LISP-System. So haben *Common LISP*-Systeme im Gegensatz zu *Scheme*-Systemen das Elementarteilchen *Funktion-cell* (↪ Abbildung 2.20 S. 249). Im Mittelpunkt dieses Abschnittes steht das Elementarteilchen *PRINT*-Name.

Wird ein Symbol eingelesen, dann konvertiert ein klassisches LISP-System in der *READ*-Phase des *READ-EVAL-PRINT*-Zyklus (↪ Abschnitt 1.1.2 S. 20) alle kleinen Buchstaben in große Buchstaben. So wird z. B. aus `FoO` oder `fOo` jeweils FOO. Dies ist ein Kennzeichen für das Alter von LISP. Als LISP Ende der 50er Jahre konzipiert wurde, konnten die Terminals (z. B. einfache Kartenleser und Drucker) nur mit Großbuchstaben umgehen. Als Terminals mit Kleinbuchstaben auf den Markt kamen, wurden Betriebssysteme und Programmiersprachen zunächst nur leicht modifiziert. Sie setzten die Kleinbuchstaben in Großbuchstaben um und konnten damit die Verarbeitung wie bisher durchführen. Heute ist bei modernen LISP-Systemen, so auch bei *PLT-Scheme* (DrScheme), diese Konvertierung standardmäßig ausgeschaltet.

Die externe Repräsentation eines Symbols ergibt sich aus der Zeichenfolge im „*Slot*“ `PRINT-name`. Dort wollen wir möglichst alle be-



Legende:
Dargestellt sind die Elementarteilchen bei einem *Common LISP*-System. Das Scheme-Konzept verzichtet bewusst auf die *Function-cell* und gewinnt dadurch an Stringenz.

`PRINT`-name	≡ Identifizierungs-Element des Symbols
`Value-cell`	≡ Speicher für (den Zeiger auf) den Symbolwert
`Funtion-cell`	≡ Speicher für (den Zeiger auf) den Funktionswert eines Symbols
`P-List`	≡ Speicher für (den Zeiger auf) die Eigenschaftsliste (*Property List*, ↪ Abschnitt 2.2.3 S. 191).
`Package-cell`	≡ Speicher für (den Zeiger auf) die Zuordnung zu einer eigenen Symbolsammlung (z. B. System, Benutzer, Editor, etc.)

Abbildung 2.20: Elementarteilchen eines Symbols (Literalatoms)

liebigen Zeichen ablegen können, z. B. auch Sonderzeichen wie Leerzeichen, öffnende oder schließende Klammern. Dazu müssen wir verhindern, dass solche Sonderzeichen in der *READ*-Phase entsprechend ihrer üblichen Aufgabe interpretiert werden. Damit solche Zeichen als Teil des Symbolnamens behandelt werden, sind sie durch ein Fluchtsymbol zu kennzeichnen. Die Zeichen „|“ (ASCII 124) und „\“(ASCII 92) dienen in Scheme als Fluchtsymbole. Das Zeichen | wird mehrfaches Fluchtsymbol genannt. Das Zeichen \ ist eine einfaches Fluchtsymbol. Beim mehrfachen Fluchtsymbol werden vom ersten Zeichen | bis zum nächsten Zeichen | alle Zeichen vom *READ*-Prozess als Symbolname interpretiert (↪ Abschnitt 3.1.1 S. 377).

```
eval> (define |(| 5) ==> |(|
eval> (define |soziale Marktwirtschaft| '\( )
eval> |soziale Marktwirtschaft| ==> |(|
eval> (eval |soziale Marktwirtschaft|) ==> 5
```

Beispiel: Mustervergleich mit Variablen für Listen

Wir betrachten erneut das `Simple-Match`-Konstrukt (↪ Abschnitt 2.4.1 S. 239) für Listen und erweitern dieses um sogenannte Mustervariablen (engl.: *match variables*). Das erweiterte Konstrukt nennen wir `Muster-vergleich`. Wir vereinbaren, dass Symbole mit einem Fragezeichen oder einem Stern zu Beginn ihres *PRINT*-Namens Mustervariablen sind. Solche Mustervariablen decken Elemente im Prüfling ab. Dabei wirken

sie wie die Symbole `?` bzw. `*`. Passt das Muster, dann sind die abgedeckten Werte durch die Mustervariablen verfügbar. Wir binden unter der Eigenschaft `Match-Value` den von der Mustervariablen abgedeckten Wert an ein namensähnliches Symbol, dessen *PRINT*-Name nicht das Kennzeichen `?` bzw. `*` aufweist. Da der Stern für mehrere Symbole im Prüfling steht, fassen wir unter der Eigenschaft `Match-Value` alle abgedeckten Elemente zu einer Liste zusammen.
Das folgende Beispiel hat die beiden Mustervariablen `?Wer` und `*Was`.

```
eval> (Mustervergleich '(?Wer begruenden *Was)
       '(Regeln begruenden einen Stil
         der Programmierung))
      ==> #t
eval> (getprop 'Wer 'Match-Value) ==> Regeln
eval> (getprop 'Was 'Match-Value)
      ==> (einen Stil der Programmierung)
```

Wie das Beispiel verdeutlicht, ist aus dem Namen einer Mustervariablen der Name des zugehörenden Symbols für die Wertebindung zu selektieren. So ist aus der Mustervariablen `?Wer` der Teil `Wer` und aus der Mustervariablen `*Was` der Teil `Was` zu ermitteln. Dazu verwenden wir das folgende `Explode`-Konstrukt[13]. Es hat als Wert eine Liste von Einzeichensymbolen, die den *PRINT*-Namen seines Arguments repräsentieren. Das Gegenstück zum `Explode`-Konstrukt ist das `Implode`-Konstrukt. Es verlangt eine Liste und baut daraus den *PRINT*-Namen des Symbols. Diese Konstrukte basieren auf den Konstrukten `symbol->string` und `string->symbol`

```
eval> (define Explode
        (lambda (Symbol)
          (letrec
              ((String (symbol->string Symbol))
               (Laenge (string-length String))
               (Sprengen
                (lambda (String Position)
                  (cond ((= Laenge 0) #f)
                        ((= Laenge Position) (list))
                        ((<= (+ Position 1) Laenge)
                         (cons (string->symbol
                                (substring String
                                  Position
                                  (+ Position 1)))
                               (Sprengen
                                 String
                                 (+ Position 1))))))))
            (Sprengen String 0))))

eval> (define Implode
```

[13] In einigen LISP-System steht es direkt zur Verfügung, z. B. in *PC Scheme*.

```
(lambda (Liste)
  (cond ((= 1 (length Liste))
         (car Liste))
        (#t (string->symbol
             (string-append
              (symbol->string
                (car Liste))
              (symbol->string
               (Implode (cdr Liste)))))))))
```

Die folgenden Beispiele zeigen die Umsetzung der *PRINT*-Repräsentation in die Elemente einer Liste bzw. ihre Zusammenfassung zu einem Symbol.

```
eval> (Explode '*Wer) ==> (* W e r)
eval> (Explode '?Was) ==> (? W a s)
eval> (Implode '(a b c)) ==> abc
eval> (Explode '|a b c|) ==> (a | | b | | c)
eval> (Implode (Explode '|a b c|)) ==> |a b c|
```

Das Konstrukt `Mustervergleich` arbeitet sowohl mit Mustervariablen als auch mit den einfachen Deckelementen Fragezeichen und Stern. Es bietet sich daher an, in einem ersten Schritt die Unterschiede in der Notation im Muster in eine einheitliche Notation zu überführen. Dazu definieren wir ein Konstrukt `Analyse-Literal-Atom`. Eine Mustervariable bzw. ein einfaches Deckelement wird in eine zweielementige Liste transformiert. Ihr erstes Element (ein Fragezeichen bzw. ein Stern) gibt den Typ an. Im Fall der Mustervariablen ist das zweite Element das Symbol, dessen Eigenschaft `MATCH-VALUE` den abgedeckten Wert aufnimmt. Liegt keine Mustervariable vor, sondern nur ein Fragezeichen bzw. ein Stern, dann ist das zweite Element `null`.

```
eval>(Analyse-Literal-Atom
       '(? Programm * bedingt *X und *Y))
 ==> ((? ()) Programm (* ()) bedingt
                     (* X) und (* Y))
```

Mit der Annahme, dass das Eingabe-Muster eine Liste ist, deren Elemente alle Symbole sind, können wir das Transformations-Konstrukt, wie folgt, definieren:

```
eval> (define Analyse-Literal-Atom (lambda (Liste)
    (cond((null? Liste) (list))
         ((symbol? (car Liste))
           (let* ((Alle-Zeichen
                   (Explode (car Liste)))
                  (Match-Typ
                   (car Alle-Zeichen))
                  (Match-Variable
                   (cond((null?
                          (cdr Alle-Zeichen))
                                      (list))
                        (#t (Implode
```

```
                    (cdr Alle-Zeichen))))))
               (cond((eq? '? Match-Typ)
                     (cons (list '?
                                 Match-Variable)
                           (Analyse-Literal-Atom
                              (cdr Liste))))
                    ((eq? '* Match-Typ)
                       (cons (list '*
                                   Match-Variable)
                             (Analyse-Literal-Atom
                               (cdr Liste))))
                    (#t (cons (car Liste)
                              (Analyse-Literal-Atom
                                   (cdr Liste)))))))
          (#t (display "Nicht implementiert: ")
              (display (car liste))))))
```

Auf der Basis dieser Eingabe-Transformation sind Prädikate zu definieren, die erkennen, ob ein Deckelement bzw. eine bestimmte Mustervariable vorliegt. Wir definieren z. B. für den Stern bzw. für die Mustervariable, deren *PRINT*-Name mit einem Stern beginnt, folgende Prädikate:

```
eval> (define *-Symbol?
        (lambda (Liste)
          (and
           (pair? Liste)
           (eq? '* (car Liste))
           (null? (list-ref Liste 1)))))
eval> (define *-Variable?
        (lambda (Liste)
          (and
           (pair? Liste)
           (eq? '* (car Liste))
           (list-ref Liste 1))))
```

Wir benötigen Selektoren und Mutatoren für den Wert einer Mustervariablen. Wir verwenden hier die Eigenschaftsliste des Symbols (↪ Abschnitt 2.2.3 S. 191), um abgedeckte Elemente zu speichern. Die Entscheidung für die Eigenschaftsliste verhindert einerseits das Überschreiben bestehender Symbol-Wert-Bindungen und dient andererseits zum Training der P-Listen-Konstrukte. Die Selektoren und den Mutator definieren wir wie folgt:

```
eval> (define Get-Match-Variable
        (lambda (Liste)
          (list-ref Liste 1)))
eval> (define Get-Match-Value
        (lambda (Symbol)
          (getprop Symbol 'Match-Value)))
eval> (define Set-Match-Value!
        (lambda (Symbol Wert)
```

```
          (putprop Symbol 'Match-Value Wert)
          #t))
```

Die Erweiterung um Mustervariablen bedingt einige Anpassungen am obigen `Simple-Match`-Konstrukt (↪ Abschnitt 2.4.1 S. 239). Zur Erläuterung sind die betroffenen Änderungsstellen in der folgenden Wiederholung des `Simple-Match`-Konstruktes markiert.

```
eval> (define Simple-Match (lambda (Muster Liste)
        (cond((and (null? Muster)
                   (null? Liste)) #t)
             ((and (null? Liste)
                   (null? (cdr Muster))
                   (eq? '* (car Muster)))
                                  #t)
                ;Ergänzung A
                ; *-Varibale? als letztes
                ; Musterelement, dann Wert-
                ; bindung () vollziehen.
              ((or (null? Muster)
                   (null? Liste)) #f)
              ((or (eq? (car Muster)
                        (car Liste))
                   (eq? '? (car Muster)))
                   (Simple-Match
                     (cdr Muster) (cdr Liste)))
                ;Ergänzung B
                ; ?-Variable? auswerten
              ((eq? '* (car Muster))
               (cond((Simple-Match
                        (cdr Muster)
                        (cdr Liste)) #t)
                    ((Simple-Match
                       Muster
                      (cdr Liste)) #t)
                    ((Simple-Match
                       (cdr Muster)
                       Liste) #t)
                    (#t #f)))
                ;Ergänzung C
                ; *-Variable? auswerten
              (#t #f))))
```

Die Ergänzung `A` betrifft den Fall, dass der Prüfling vollständig abgearbeitet ist, d. h. die `lambda`-Variable `Liste` ist an `null` gebunden, aber das Muster enthält noch Elemente. Dann ist festzustellen, ob das Muster nur noch ein Element enthält und dieses entweder das Prädikat `*-Symbol?` oder das Prädikat `*-Variable?` erfüllt. Ist `*-Symbol?` bzw. `*-Variable?` erfüllt, dann passt das Muster, so dass wir `#t` zurückgeben können und im Fall `*-Variable?` noch das zugehörende

Symbol an `null` binden. Wir unterstellen damit, dass unser Muster auch dann passt, wenn der Stern für eine leere Folge von Elementen im Prüfling steht.

Die Ergänzung `B` betrifft den Fall, dass im Muster das Prädikat `?-Variable?` erfüllt ist. Hierzu ist eine entsprechende Klausel einzufügen. Wenn der Rest des Musters mit dem Rest des Prüflings passt, ist das zugehörende Symbol an den Wert zu binden. Wir können diese Klausel, wie folgt, definieren:

```
((and  (?-Varaible? (car Muster))
       (Match (cdr Muster) (cdr Liste)))
   (Set-Match-Value!
     (Get-Match-Variable (car Muster))
                  (car Liste)) #t)
```

Die Ergänzung `C` betrifft den Fall, dass im Muster das Prädikat `*-Variable?` erfüllt ist. Im `Simple-Match` ist quasi der Fall formuliert, der `*-Symbol?` abdeckt. Statt `(eq?'* (car Muster))` ist `(*-Symbol? (car Muster))` zu setzen. Die einzelnen Klauseln dieses `*-Symbol?`-Falles sind hier für den Fall `*-Variable?` ebenfalls zu formulieren, allerdings sind dann die Wertbindungen zu ergänzen. Wir können diese Ergänzung, wie folgt, definieren:

```
((*-Variable? (car Muster))
   (cond( ;;*-Variable? deckt ein Element!
          (Match (cdr Muster) (cdr Liste))
           (Set-Match-Value!
            (Get-Match-Varaible (car Muster))
                                (list
                                  (car Liste))))
          ;;*-Variable? deckt mehrere Elemente!
          ;; Der bisherige Wert ist zu ergänzen.
         ((Match Muster (cdr Liste))
            (Set-Match-Value!
             (Get-Match-Variable (car Muster))
              (cons (car Liste)
                (Get-Match-Value
                  (Get-Match-Variable
                   (car Muster)))))
             #t)
          ;;*-Variable? deckt kein Element!
         ((Match (cdr Muster) Liste) #t)
          ;; Muster passt nicht.
         (#t #f)))
```

Dieses erweiterte `Match`-Konstrukt und seine Hilfsfunktionen wie Prädikate, Selektoren, Mutatoren sowie das `Analyse-Literal-Atom` sind als lokale Funktionen im Konstrukt `Mustervergleich` definiert (↪ Programm S. 256).

Zum Verstehen der Aufgabe sind vorab Anforderungen, Annahmen zum Entwurf und Testfälle formuliert. Dabei verweist die Abkürzung `A` auf eine *Anforderung*, die Abkürzung `E` auf eine Annahme zum *Entwurf* und die Abkürzung `T` auf einen *Testfall*.

`A1:` `Mustervergleich` analysiert zwei Listen, die Liste `Pattern` und die Liste `Objekt`, mit dem Ziel Werte für Match-Variable zu bestimmen.

`A2:` `Pattern` kann aus folgenden Symbolen bestehen:

- `A2.1:` `?` ≡ Deckt ein Listenelement im `Objekt`.
- `A2.2:` `*` ≡ Deckt eine Folge von Listenelementen im `Objekt`.
- `A2.3:` `?`< *name* > ≡ Deckt ein Listenelement im `Objekt` und bindet dieses Listenelement unter der Eigenschaft `Match-Value` an das Symbol < *name* >.
- `A2.4:` `*`< *name* > ≡ Deckt eine Folge von Listenelementen im `Objekt` und bindet diese Folge als Liste unter der Eigenschaft `Match-Value` an das Symbol < *name* >.
- `A2.5:` < *symbol* > ≡ Deckt das gleiche < *symbol* > im `Objekt`.

`E1:` Konvertierung der Matchvariablen in eine Liste mit 2 Elementen, wie folgt:

```
E1.1: ?        ==> (? ())
E1.2: *        ==> (* ())
E1.3: ?<name> ==> (? <name>)
E1.3: *<name> ==> (* <name>)
```

`T1:` Beispiel

```
eval> (Mustervergleich '(*X hat ? wartbare ?Y)
      '(Der Entwurf hat eine wartbare Struktur))
      ==> #t
eval> (getprop 'X 'Match-Value) ==> (Der Entwurf)
eval> (getprop 'Y 'Match-Value) ==> Struktur
```

Programm: Mustervergleich mit Mustervariablen (Abbildungsbasis: Listen)

```
eval> (define Mustervergleich (lambda (Pattern Object)
  (letrec (
      ;;Hilfsfunktion Symbol in Liste zerlegen
      (Explode
        (lambda (Symbol)
          (letrec
              ((String (symbol->string Symbol))
               (Laenge (string-length String))
               (Sprengen
                (lambda (String Position)
                  (cond ((= Laenge 0) #f)
                        ((= Laenge Position) (list))
                        ((<= (+ Position 1) Laenge)
                         (cons (string->symbol
                                (substring
                                  String
                                  Position
                                 (+ Position 1)))
                               (Sprengen
                                 String
                                 (+ Position 1)
                               )))))))
        (Sprengen String 0))))
       ;;Hilfsfunktion aus Liste Symbol erzeugen
      (Implode
        (lambda (Liste)
          (cond ((= 1 (length Liste))
                   (car Liste))
                (#t (string->symbol
                     (string-append
                      (symbol->string
                        (car Liste))
                      (symbol->string
                       (Implode (cdr Liste)))))))))
      ;;Konvertierung der Matchvariablen in eine
      ;; zweielementige Liste
      (Analyse-Literal-Atom (lambda (Liste)
       (cond((null? Liste) (list))
            ((symbol? (car Liste))
              (let* ((Alle-Zeichen
                        (Explode (car Liste)))
                     (Match-Typ
                       (car Alle-Zeichen))
                     (Match-Variable
                       (cond((null?
                              (cdr Alle-Zeichen))
                             (list))
                            (#t (Implode
                                  (cdr
                                    Alle-Zeichen)
                                 )))))
                (cond((eq? '? Match-Typ)
```

```
                          (cons
                            (list '? Match-Variable)
                              (Analyse-Literal-Atom
                               (cdr Liste))))
                         ((eq? '* Match-Typ)
                          (cons
                            (list '* Match-Variable)
                              (Analyse-Literal-Atom
                               (cdr Liste))))
                         (#t (cons
                               (car Liste)
                                 (Analyse-Literal-Atom
                                   (cdr Liste)))))))
              (#t (display "Nicht implementiert: ")
                  (display (car Liste))))))
;;;;Prädikate
      (?-Symbol? (lambda (x)
         (and (pair? x)
              (eq? '? (car x))
              (null? (list-ref x 1)))))
      (?-Variable? (lambda (x)
         (and (pair? x) (eq? '? (car x))
                        (list-ref x 1))))
      (*-Symbol? (lambda (x)
         (and (pair? x)
              (eq? '* (car x))
              (null? (list-ref x 1)))))
      (*-Variable? (lambda (x)
         (and (pair? x) (eq? '* (car x))
                        (list-ref x 1))))
;;;;Selektoren
      (Get-Match-Variable
        (lambda (x) (list-ref x 1)))
      (Get-Match-Value
        (lambda (symbol)
         (getprop symbol 'Match-Value)))
;;;;Mutator
      (Set-Match-Value!
        (lambda (symbol wert)
         (putprop symbol 'Match-Value wert)
         #t))
;;;;Match-konstrukt
      (Match (lambda (Muster Liste)
         (cond((and (null? Muster)
                    (null? Liste)) #t)
               ((null? Liste)
               (cond((and (*-Symbol?
                            (car Muster))
                          (null?
                           (cdr Muster))) #t)
                     ((and (*-Variable?
                            (car Muster))
                           (null?
                             (cdr Muster)))
                      (Set-Match-Value!
                         (Get-Match-Variable
```

```
                          (car Muster))
                        null)
                  #t)
                 (#t #f)))
           ((or (equal? (car Muster)
                        (car Liste))
                (?-Symbol? (car Muster)))
            (Match (cdr Muster) (cdr Liste)))
           ((and (?-Variable? (car Muster))
                 (Match (cdr Muster)
                  (cdr Liste)))
            (Set-Match-Value!
               (Get-Match-Variable
                 (car Muster))
               (car Liste))
            #t)
           ((*-Symbol? (car Muster))
            (cond((Match (cdr Muster)
                         (cdr Liste)) #t)
                 ((Match Muster
                         (cdr Liste)) #t)
                 ((Match (cdr Muster)
                          Liste) #t)
                 (#t #f)))
           ((*-Variable? (car Muster))
            (cond((Match (cdr Muster)
                         (cdr Liste))
                  (Set-Match-Value!
                     (Get-Match-Variable
                         (car Muster))
                     (list (car Liste))))
                 ((Match Muster (cdr Liste))
                  (Set-Match-Value!
                     (Get-Match-Variable
                       (car Muster))
                     (cons
                      (car Liste)
                       (Get-Match-Value
                        (Get-Match-Variable
                         (car Muster)))))
                  #t)
                 ((Match
                   (cdr Muster) Liste) #t)
                 (#t #f)))
           (#t #f)))))
   (Match (Analyse-Literal-Atom Pattern)
          Object))))
;;;Applikation
eval> (Mustervergleich '(*X hat ? wartbare ?Y)
         '(Der Entwurf hat eine wartbare Struktur))
      ==> #t
eval> (getprop 'X 'Match-Value)
      ==> (Der Entwurf)
eval> (getprop 'Y 'Match-Value)
      ==> Struktur
```

2.4.4 Generieren eines Symbols

Wird der Name eines Symbol eingelesen, dann ist zu prüfen, ob dieses Symbol schon existiert oder nicht. Da wir laufend mit Symbolen umgehen, ist so effizient wie möglich zu prüfen, ob es sich um ein vorhandenes oder neues, zu erzeugendes Symbol handelt. Z. B. benötigt das `define`-Konstrukt das Prüfergebnis, um zu entscheiden, ob nur der Wert eines existierenden Symbols zu ändern oder zunächst ein entsprechendes Symbol zu erzeugen und dann in der Umgebung mit seinem Wert einzutragen ist.

```
eval> Foo ==> ERROR ...
       ;In der Umgebung nicht definiert
       ; ... reference to an identifier
       ; before its definition: Foo
eval> (define Foo 6) ;Foo generiert
                     ; und Wert zugewiesen.
eval> (define Foo '(A B C)) ;Foo neuen
                            ; Wert zugewiesen.
eval> Foo ==> (A B C)
```

Diese Prüfung nimmt üblicherweise Bezug auf eine Symboltabelle. Dabei werden neue Symbole erzeugt und in die Tabelle eingetragen. Diesen Vorgang nennt man auch „*Internmachen*“. Über das Einlesen des Namens des Symbols können daher nur intern gemachte Symbole wieder erreicht werden.

Erzeugt man ein Symbol, ohne es „intern zu machen“, d. h. ohne Eintragung in die Symboltabelle, dann ist es später durch die *READ*-Phase nicht erreichbar. Obwohl ein „nicht interngemachtes“ Symbol über das Einlesen seines Namens nicht referenzierbar ist, kann es sehr nützlich sein. Es ist keinesfalls mit Symbolen zu verwechseln, die überhaupt nicht mehr erreichbar sind und als Platzverschwendung (Müll) zu betrachten sind. Erreichbar ist ein solches „nicht internes“ Symbol (engl.: *uninterned symbol*) z. B. wenn es als Wert eines internen Symbols dient.

Ein nicht „interngemachtes“ Symbol generiert das `gensym`-Konstrukt. Ein von `gensym` erzeugtes Symbol hat keine Gleichheit (im Sinne von `eq?`) mit existierenden oder neu eingelesenen Symbolen. Geht es daher um die Vermeidung von Überschreibungen mit bestehenden oder noch entstehenden Symbolen, dann ist das `gensym`-Konstrukt angebracht.

`eval>` (`gensym` $\{<$ *argument* $>\}$) `==>` $<$ *uninterned* $-$ *symbol* $>$

mit:

$<$ *argument* $>$	$\equiv <$ *symbol* $> \mid <$ *string* $>$
$<$ *symbol* $>$	$\equiv$ Der Symbolwert wird als Suffix (Anhang) des Symbolnamens verwendet. Ist kein $<$ *symbol* $>$ bzw. kein $<$ *string* $>$ angegeben beginnt sein *PRINT*-Name mit `g` gefolgt von einer laufenden Nummer. Jede `gensym`-Applikation erhöht die laufende Nummer.
$<$ *string* $>$	$\equiv$ Die Zeichenkette wird als Suffix (Anhang) des Symbolnamens verwendet; $\hookrightarrow <$ *symbol* $>$.
$<$ *uninterned* $-$ *symbol* $>$	$\equiv$ Ein neues nicht intern gemachtes Symbol.

```
eval> (define Foo (gensym))
eval> Foo ==> g754
eval> (define Bar (gensym))
eval> Bar ==> g755
eval> (symbol? Foo) ==> #t
eval> (eq? Foo 'g754)
      ==> #f  ;Achtung!
              ; Keine Gleichheit trotz
              ; gleichen PRINT-Namens
eval> (eq? Foo Foo) ==> #t
eval> (define Baz (gensym "754"))
eval> Baz ==> |754756|
eval> (define Foo (gensym Baz))
eval> Foo ==> |754756757|
```

Wollen wir ein „nicht interngemachtes“ Symbol mit vorgegebenen Namen generieren, dann ist das `string->uninterned-symbol`-Konstrukt zu verwenden.

```
eval> (define Foo
        (string->uninterned-symbol
          "M O R E"))
eval> Foo ==> |M O R E|
eval> (eq? Foo '|M O R E|) ==> #f
```

Im Vergleich dazu, zeigt das folgende Beispiel den Effekt des „Interngemachens“:

```
eval> (define Foo (string->symbol "M O R E"))
eval> Foo ==> |M O R E|
eval> (eq? Foo '|M O R E|) ==> #t
```

Beispiel: Pseudo-Code-Vorübersetzer

In Kapitel 1 ist für die Alternative und die Iteration eine Notation angegeben, die häufig in den ersten Phasen des Lebenszyklus eines Software-Produktes (↪ Abbilung 1.13 S. 51) eingesetzt wird. Man spricht in diesem Zusammenhang von einer Pseudocode-Notation, weil die formulierten Konstrukte prinzipiell noch nicht „lauffähig“ sind. Sie enthalten

Formulierungen, die später noch zu präzisieren sind. Wir fassen hier jedoch die Pseudo-Code-Notation nur als eine syntaktische Erleichterung auf und unterstellen eine „lauffähige“ Notation.

Wir definieren mit Hilfe des Konstruktes `Mustervergleich` einen Pseudocode-Vorübersetzer. Unser `P-Code`-Konstrukt kann folgende Pseudocode-Notationen in LISP-Konstrukte übersetzen:

- Alternative (↪ S. 80)
  `(if ... then ... else ... fi)`
  `(if ... then ... fi)`

- Iteration (↪ S. 89)
  `(while ... do ... od)`
  `(do ... until ... od)`

Diese Pseudocode-Angaben betrachten wir als eine Musterbeschreibung. Wir formulieren z. B. für die Alternative das Muster `(if ?P then *Ja else *Nein)`. Passt das Muster, dann sind die Mustervariablen an die aktuellen Werte gebunden. Unser `P-Code`-Konstrukt braucht nur die `Ja`- und `Nein`-Werte an die entsprechenden Stellen des geeigneten LISP-Konstruktes zu setzen.

Da eine Mustervariable mit einem Stern die abgedeckten Elemente zu einer Liste zusammenfasst, befindet sich das erste abgedeckte Element in dieser Liste in funktionaler Stellung. Es würde als Operator interpretiert. Dies verhindern wir durch das Einfügen des `begin`-Konstruktes in die `Ja`- bzw. `Nein`-Liste.

```
eval> (P-Code
       '(if P then A B C else D E F fi))
       ==>
      (if P (begin A B C) (begin D E F))
```

Für die beiden Iterationen formulieren wir die beiden Muster `(while ?P do *Body od)` und `(do *Body until ?Q od)`. Mit den Werten der Mustervariablen `?P` bzw. `?Q` und `*Body` konstruieren wir eine rekursive LISP-Funktion. Diese Funktion nutzt das `letrec`-Konstrukt zur Definition einer lokalen, rekursiven Funktion. Benötigt wird ein Funktionsname (Symbol) für den Selbstbezug. Jedes Symbol, das wir dafür rein zufällig wählen, kann zu ungewollten Nebeneffekten führen.

Im folgenden Beispiel haben wir unterstellt, wir hätten das Symbol `AAA` als Funktionsname gewählt. Tritt in der Pseudo-Code-Formulierung zufällig auch das Symbol `AAA` auf, dann kommt es zu Fehlern. Der erste Fall hat keine Namensgleichheit mit dem Symbol `AAA`; der zweite hat eine solche und führt zu einem ungewollten Effekt. Im 3. Fall verwenden wir ein Symbol, das mit dem Konstrukt `gensym` generiert wurde, und verhindern damit einen solchen Nebeneffekt.

Fall 1: Keine zufällige Namensgleichheit mit der Funktion `AAA`

```
eval> (define I 0)
eval> (define J 'EUR)
eval> (define Foo
        (P-Code
         '(do (display I)
              (display J)
              (newline)
              (set! I (+ I 1))
           until (> I 3) od)))
eval> Foo ==> (letrec
                  ((AAA (lambda ()
                          (display I)
                          (display J)
                          (newline)
                          (set! I (+ I 1))
                          (cond((> I 3))
                               (#t (AAA))))))
                  (AAA))
eval> (eval Foo) ==>
      0EUR
      1EUR
      2EUR
      3EUR
      #t
```

Fall 2: Zufällige Namensgleichheit mit der Funktion `AAA`

```
eval> (define I 0) ==> I
eval> (define AAA 'EUR) ;Namensgleiche
                        ; globale Variable
eval> (define Foo
        (P-Code
         '(do (display I)
              (display AAA)
              (newline)
              (set! I (+ I 1)) until (> I 3) od)))
eval> Foo ==> (letrec
                  ((AAA (lambda ()
                          (display I)
                          (display AAA)
                          (newline)
                          (set! I (+ I 1))
                          (cond((> I 3))
                                (#t (AAA))))))
                  (AAA))
eval> (eval Foo) ==>     ;Achtung!
      0#<procedure:AAA>  ; Ungewollter Nebeneffekt!
```

```
      1#<procedure:AAA>  ; Gewünschte Lösung
      2#<procedure:AAA>  ; vgl. Fall 1
      3#<procedure:AAA>
      #t
```

Im `P-Code`-Konstrukt (↪ Programm S. 264) ist das Symbol für die lokale, rekursive Funktion mit Hilfe des `gensym`-Konstruktes konstruiert. Selbst in dem ungünstigsten Fall, d. h. der *PRINT*-Name dieses generierten Symbols ist gleich mit dem *PRINT*-Namen eines eingelesenen Symbols, besteht keine Referenz zu diesem anderen Symbol. Der Fall 3 zeigt eine solche, schwer durchschaubare Situation. Zu einem ungewollten Nebeneffekt kommt es jedoch nicht.

Fall 3: Symbol für die Funktion mit `gensym` generiert

```
eval> (define I 0)
eval> (define g794 'EUR) ;Zufällige PRINT-
                         ; Namensgleichheit mit
                         ; dem anschließend
                         ; generierten Symbol.
eval> (define Foo
        (P-Code
         '(do (display I)
              (display g794)
              (newline)
              (set! I (+ I 1)) until (> I 3) od)))
eval> Foo ==> (letrec
                 ((g794  ;Generiertes Symbol
                   (lambda ()
                        (display I)
                        (display g794);Eingegebenes
                                      ; Symbol
                        (set! I (+ I 1))
                        (cond((> I 3))
                              (#t (g794));Generier-
                              ))))     ; tes Symbol

                 (g794)) ; Generiertes Symbol
eval> (eval Foo) ==>
      0EUR
      1EUR
      2EUR
      3EUR
      #t
```

Im Fall der Iteration setzt das `P-Code`-Konstrukt seinen Rückgabewert (= `letrec`-Konstrukt) mit Hilfe der wiederholten Anwendung des `list`-Konstruktes zusammen. Die geschachtelten `list`-Konstrukte sind nicht sehr übersichtlich (↪ Programm S. 264). Mit Hilfe von Makros

wäre eine syntaktisch einfachere Notation möglich. Mit Makros befassen wir uns eingehend im Abschnitt 2.5.3 S. 293.

Zum Verstehen der Aufgabe sind vorab Anforderungen und Testfälle formuliert. Dabei verweist die Abkürzung `A` auf eine *Anforderung* und die Abkürzung `T` auf einen *Testfall*.

`A1`: `P-Code` überführt sogenannte Pseudocode-Notationen in ablauffähige LISP-Konstrukte.

`A2`: Implementiert sind die Alternative (↪ S. 80):

`A2.1`: `(if ... then ... else ... fi)`

`A2.2`: `(if ... then ... fi)`

`A3`: und die Iteration (↪ S. 89):

`A3.1`: `(while ... do ... od)`

`A3.2`: `(do ... until ... od)`

`T1`: Beispiel

```
eval> (P-Code
       '(while
         (<= I 5)
         do
        (display I)
        (set! I (+ I 1))
        od))
      ==> (letrec
           ((g799 (lambda ()
                    (cond((<= I 5)
                     (display I)
                     (set! I (+ I 1))
                     (g799))
                    (#t #t)))))
                  (g799))
eval> (letrec ...) ==> 012345#t
```

Programm: `P-Code` **Vorübersetzer**

```
(define P-Code
 (lambda (Sexpr)
  (cond((Mustervergleich
        '(if ?X then *Y else *Z fi) Sexpr)
        (let ((Praedikat
               (getprop 'X 'Match-Value))
              (Body-Liste-1
               (getprop 'Y 'Match-Value))
```

```
          (Body-Liste-2
           (getprop 'Z 'Match-Value)))
      (list 'if
            Praedikat
            (cons 'begin Body-Liste-1)
            (cons 'begin Body-Liste-2))))
 ((Mustervergleich
   '(if ?X then *Y fi) Sexpr)
   (let ((Praedikat
          (getprop 'X 'Match-Value))
         (Body-Liste
          (getprop 'Y 'Match-Value)))
      (list 'if
            Praedikat
            (cons 'begin Body-Liste)
            (list))))
 ((Mustervergleich
   '(while ?X do *Y od) Sexpr)
   (let ((Symbol (gensym))
         (Praedikat
          (getprop 'X 'Match-Value))
         (Body-Liste
          (getprop 'Y 'Match-Value)))
      (list 'letrec
            (list (list Symbol
                        (list 'lambda null
                          (list 'cond
                                (append
                                  (cons
                                    Praedikat
                                    Body-Liste)
                                  (list
                                   (list Symbol)))
                                '(#t #t)))))
            (list Symbol))))
 ((Mustervergleich
   '(do *Y until ?X od) Sexpr)
   (let ((Symbol (gensym))
         (Praedikat
          (getprop 'X 'Match-Value))
         (Body-Liste
          (getprop 'Y 'Match-Value)))
      (list 'letrec
            (list (list Symbol
                        (append
                           (cons 'lambda
                                 (cons
                                   null
                                   Body-Liste))
                           (list (list 'cond
                             (list Praedikat)
                               (list '#t
                                 (list
                                   Symbol)))))))
            (list Symbol))))
(#t (display
```

```
          "Keine P-code-Transformation fuer: ")
         (display Sexpr)))))
```

Zur Verdeutlichung des `P-Code`-Programms hier einige Applikationen:

```
eval> (define Frage
        (P-Code '(if (> I 0)
           then (display "Alles klar?")
           else (display "Doch nicht?") fi)))
eval> Frage ==>
      (if (> I 0)
       (begin (display "Alles klar?"))
       (begin (display "Doch nicht?")))
eval> (define I 3)
eval> (eval Frage) ==> Alles klar?
eval> (define I 0)
eval> (eval Frage) ==> Doch nicht?
eval> (define I 3)
eval> (define Foo
        (P-Code
         '(while (< I 6)
                 do
                 (display I)
                 (newline)
                 (set! I (+ I 1))
                 od)))
eval> Foo ==>
     (letrec
      ((g873 (lambda ()
               (cond ((< I 6)
                      (display I)
                      (newline)
                      (set! I (+ I 1))
                      (g873))
                     (#t #t)))))
               (g873))
eval> (eval Foo) ==>
      3
      4
      5
      #t
```

2.4.5 Zusammenfassung: `Explode`, `Implode` und `gensym`

Die Zeichenkette (engl.: *string*) kann mit einer Vielzahl spezieller Konstrukte, z. B. `make-string`, `substring` oder `string-ref`, bearbei-

tet werden. In Bezug zur Liste tritt an die Stelle des Prädikates `null?` ein Prädikat `string-null?` auf der Basis von `string-length`. Das Prädikat `string=?` entspricht der `equal?`-Prüfung.

Ein Symbol, auch Literalatom genannt, ist eine „Einheit“ mit eigener Identität. Der *PRINT*-Name ist eines der Elementarteilchen der „Einheit“ Symbol. Er kann durch Nutzung von Fluchtzeichen (| und \) auch mit Sonderzeichen konstruiert werden. Mit Hilfe von `string->symbol` und `symbol->string` können die Konstrukte `Explode` und `Implode` definiert werden (↪ S. 250), falls das LISP-System (z. B. *PC Scheme*) diese nicht schon bereitstellt. Mit `Explode` und `Implode` ist der *PRINT*-Name als Informationsträger manipulierbar.

Ein „nicht interngemachtes“ Symbol (engl.: *uninterned symbol*) ist über den *PRINT*-Namen nicht referenzierbar. Ein neues, „nicht interngemachtes“ Symbol generiert das `gensym`-Konstrukt. Eine `gensym`-Applikation ist zweckmäßig, wenn ein Namenskonflikt mit existierenden oder später generierten Symbolen zu verhindern ist.

Im Gegensatz zu klassischen LISP-Systemen hat das Symbol in Scheme keine Funktionszellen als besonderes Elementarteilchen. Damit werden Ungereimtheiten zwischen funktionalen und nichtfunktionalen Werten vermieden. Mit dem `gensym`-Konstrukt ist auch in Scheme eine besondere Funktionszelle mit Hilfe der Eigenschaftsliste realisierbar.

```
eval> (define Get-F 0)
eval> (define Set-F! 0)
eval> (let
        ((Eigenschaft (gensym)))
        (set! Get-F
          (lambda (Symbol)
            (getprop Symbol Eigenschaft)))
        (set! Set-F!
          (lambda (Symbol Funktion)
            (putprop Symbol
                     Eigenschaft
                     Funktion)))
      #t)
      ==> #t
eval> (Set-F! 'Foo
              (lambda (x y)
               (+ x y)))
eval> (define Foo "Alles klar?")
eval> Foo ==> "Alles klar?"
eval> ((Get-F 'Foo) 4 5) ==> 9
```

Charakteristische Beispiele für Abschnitt 2.4

```
eval> ;;;;Pascalsches Dreieck ausgeben.
      (define |Pascalsches Dreieck|
        (lambda (n_Endzeile)
```

```
(letrec
   ((Pascal
     (lambda (Grundseite i_Zeile)
       (cond((= n_Endzeile i_Zeile)
               (Ausgabe Grundseite))
             (#t (Ausgabe Grundseite)
                (Pascal
                   (Make-neue-Dreieck-Seite
                     Grundseite)
                   (+ i_Zeile 1))))))
    (Make-neue-Dreieck-Seite
     (lambda (l)
       (cond((= 1 (length l)) (list 1 1))
             (#t (append (list 1)
                   (append (Summe  l)
                     (list 1)))))))
    (Summe
     (lambda (l_zahlen)
       (cond((= (length l_zahlen) 2)
              (list (+ (car l_zahlen)
                       (cadr l_zahlen))))
             (#t (append
                   (list (+ (car l_zahlen)
                            (cadr l_zahlen)))
                   (Summe (cdr l_zahlen)))))))
    (Laenge-Ausgabe-Zeile 70)
    (Anzahl-Ziffern-Grundseite
     (lambda (l)
       (cond((null? l) 0)
              ;;auf Zahlen angewendet,
              ;; erst in Symbol konvertieren!
             (#t (+ (length
                      (Explode
                       (string->symbol
                        (number->string
                         (car l)))))
                    (Anzahl-Ziffern-Grundseite
                      (cdr l))))))))
    (Anzahl-Space-Grundseite
     (lambda (l)
       (- (length l) 1)))
    (Einruecken-Ausgabe-Zeile
     (lambda (l)
       (make-string
          (floor
             (/ (- Laenge-Ausgabe-Zeile
                   (+ (Anzahl-Ziffern-Grundseite
                       l)
                      (Anzahl-Space-Grundseite
```

```
                                    l)))
                             2))
                        #\space)))
                (Ausgabe
                 (lambda (l)
                   (display
                    (Einruecken-Ausgabe-Zeile l))
                   (display l)
                   (newline))))
              (Pascal (list 1) 0))))
eval> (|Pascalsches Dreieck| 12) ==>
                              (1)
                             (1 1)
                            (1 2 1)
                           (1 3 3 1)
                          (1 4 6 4 1)
                        (1 5 10 10 5 1)
                       (1 6 15 20 15 6 1)
                     (1 7 21 35 35 21 7 1)
                   (1 8 28 56 70 56 28 8 1)
                 (1 9 36 84 126 126 84 36 9 1)
             (1 10 45 120 210 252 210 120 45 10 1)
           (1 11 55 165 330 462 462 330 165 55 11 1)
         (1 12 66 220 495 792 924 792 495 220 66 12 1)

eval> ;;;;Vergleich zweier PRINT-Namen
      (define |x <= y|
         (lambda (x y)
         (let
           ((x-Symbol (Explode x))
            (y-Symbol (Explode y)))
         (do
             ((x-Sym x-Symbol (cdr x-Sym))
              (y-Sym y-Symbol (cdr y-Sym))
              (Wert #t))
           ((or (not Wert)
                (null? x-Sym)) Wert)
           (if (and
                y-Sym
                (string<=?
                    (symbol->string (car x-Sym))
                    (symbol->string (car y-Sym))))
                (set! Wert #t)
                (set! Wert #f))))))
eva> (|x <= y| 'Dick 'Dic)
     ==> ERROR ...
         ;car: expects argument of
         ; type <pair>; given ()
eval> (|x <= y| 'Dick 'Dick)
```

```
      ==> #t
eval> (|x <= y| 'Dick 'DicKe)
      ==> #f
eval> (gensym) ==> g876
eval> (|x <= y| 'g877 (gensym))
      ==> #t
```

2.5 Abbildungsoption: Funktion mit Umgebung

Der klassische LISP-Kern konzipiert von *John McCarthy* (↪ z. B. [124]), das sogenannte „*Pure LISP*“, ist eine applikative Sprache, da die Applikation (Anwendung) von Funktionen das wesentliche Konstruktionsmittel ist. Dieses „*Pure LISP*“ ist gleichzeitig eine funktionale Sprache, da die Abstraktion auf der Komposition von Funktionen (≡ evaluierten `lambda`-Konstrukten) beruht. Diese Schachtelung ermöglicht das Kombinieren primitiver Konstrukte zu komplexen anwendungsspezifischen Lösungen.

„*Pure LISP*“ ist der Stammvater vieler Sprachen für das funktionale Programmieren (↪ z. B. [87, 46, 89, 55]). Zu nennen sind z. B. *FP* (↪ [7]), *HOPE* (genannt nach „*Hope Park Square*“, Adresse von Edinburgh's Department of Computer Science) oder *ML* (↪ [75]). Leitidee der funktionalen Programmierung ist der Funktionswert, der nur durch Werte der Argumente bestimmt wird. So oft wir auch eine Funktion mit den gleichen Argumenten aufrufen, stets hat sie den gleichen Wert. Diese Eigenschaft (engl.: *referential transparency*) erleichtert wesentlich das Durchschauen einer Konstruktion. Beschränken wir uns auf Konstrukte mit dieser Eigenschaft, dann ist eine Funktion problemlos durch eine effizientere Funktion, die den gleichen Wert bei gleichen Argumenten hat, austauschbar. Diesen Vorteil gibt es nicht kostenlos. Solche nebeneffektfreien Konstrukte erstellen neue Objekte statt modifizierte Versionen von ihren Argumenten. Die Neukonstruktion hat im Regelfall einen höheren Rechenzeit- und Speicherbedarf.

Stets ist die Funktion im Zusammenhang mit ihrer Umgebung zu sehen. Erst diese Verknüpfung zum sogenannten *Closure*-Konzept meistert Funktionen höherer Ordnung. Die folgende Definition, in *TLC-LISP* formuliert, zeigt explizit die Verknüpfung der Umgebung (ENVironemt) mit dem `LAMBDA`-Konstrukt durch das explizit notierte `CLOSURE`-Konstrukt.

```
T-eval> (DE ADDN (N)
          ;;"Abschließung" für
          ;; das folgende lambda-Konstrukt
          (CLOSURE
            ;; Hier ist N eine freie Variable
            (LAMBDA (M) (ADD N M))
            ;;Umgebung speichert Variable N
            (ENV 'N N)))
```

```
          ==> ADDN
T-eval> (SETQ PLUS3 (ADDN 3))
        ==> (CLOSURE (lambda (M)
                       (ADD N M))
                     (ENV 'N 3))
T-eval> (PLUS3 7) ==> 10
```

Die Bindungsstrategie bestimmt, zu welchem Zeitpunkt die Variablen ihren Wert erhalten. Die jeweilige Implementation bestimmt, auf welche Art und Weise der Wert einer Variablen gespeichert und nachgeschaut wird. Bei der ersten Erörterung der Umgebung (↪ S. 42) haben wir die *dynamische* und die *lexikalische* Bindungsstrategie angedeutet. Die beiden wesentlichen Implementationskonzepte werden als „*deep access*“ und „*shallow access*“ gekennzeichnet. Die Implementation *deep access* hat im Vergleich zu *shallow access* den Vorteil ohne großen Aufwand den Wechsel der aktuellen Umgebung vollziehen zu können. *Shallow access* hat demgegenüber den Vorteil auf den Wert der Variablen schnell zugreifen zu können. Im folgenden werden diese Implementationskonzepte nicht weiter diskutiert (Näheres dazu ↪ z. B. [2]).

Wir unterstellen die lexikalische Bindungsstrategie und behandeln damit die Funktion und ihre Umgebung als Option, lokale Werte abbilden zu können. Zunächst vertiefen wir Funktionen höherer Ordnung (↪ Abschnitt 2.5.1 S. 271).

Funktionen höherer Ordnung können Funktionen als Werte von `lambda`-Variablen haben. Dabei zeigen wir, dass es sinnvoll sein kann, sogar die Funktion selbst an ihre `lambda`-Variable zu binden. Mit dem allgemeinen Kontrollstrukturkonstrukt `call-with-current-continuation` lässt sich an die `lambda`-Variable die Fortsetzung der Verarbeitung, d. h. die Restkonstruktion, binden. Das *Continuation*-Konzept von Scheme ermöglicht damit beliebige Fortsetzungen einer begonnen Abarbeitung. Es gestattet quasi Sprünge in ihre „Zukunft“ und ihre „Vergangenheit“. Wir erläutern diese allgemeine Kontrollstruktur und behandeln den Unterschied zu einem einem einfachen *GOTO*-Sprung (↪ Abschnitt 2.5.2 S. 285).

Eine hierarchische Ordnung von Umgebungen ermöglicht das Arbeiten mit abgegrenzten Paketen. Als Beispiel dient erneut die Kontenverwaltung (↪ Abschnitt 2.7 S. 319). Dabei werden Makros als syntaktische Verbesserungen genutzt. Wir befassen uns daher vorab mit der Definition von Makros (↪ Abschnitt 2.5.3 S. 293).

2.5.1 Funktionen höherer Ordnung

In diesem Abschnitt vertiefen wir das Verständnis für Funktionen, die Funktionen als Argumente und/oder als Wert haben. Das Konstruieren mit solchen Funktionen höherer Ordnung (engl.: *higher-order function*) wird anhand von drei Beispielen erörtert. Das erste Beispiel bil-

det den Listen-Konstruktor `cons` als Funktion ab (↪ Abschnitt 2.5.1 S. 272). Das zweite Beispiel definiert *Y*, den Fixpunktoperator für Funktionen. Die *Y*-Definition hat hier die Aufgabe, das Verständnis für rekursive Funktionen zu vertiefen. Sie ermöglicht die Definition rekursiver Funktionen ohne Verwendung des `define`- oder `letrec`Konstruktes (↪ Abschnitt 2.5.1 S. 277).

Funktionale Abbildung der `cons`-Zelle

Die funktionale `cons`-Abbildung nennen wir `F-cons`, um eine Namensgleichheit und infolgedessen das Überschreiben des eingebauten `cons`-Konstruktes zu vermeiden. Zu `F-cons` definieren wir die Selektoren `F-car`, `F-cdr`, die Prädikate `F-pair?`, `F-null?` und die Mutatoren `F-set-car!` und `F-set-cdr!`.

Die Konstruktionsbasis („Mutter Erde“) einer Liste ist `null`, die leere Liste. Analog dazu ist eine funktionale Anfangsgröße, ein funktionales `null`, zu definieren. Wir nennen es `*F-NULL*`. Die Sterne im Namen dienen dazu, eine versehentliche Namensgleichheit beim Programmieren zu erschweren, weil wir nur wichtigen globalen Variablen einen Namen mit Sternen geben (↪ Abschnitt 3.1.1 S. 377).

```
eval> (define *F-NULL*
        (lambda()
          )) ==> ERROR ... ;Kein Funktionskörper
```

Das `lambda`-Konstrukt erfordert zumindest einen symbolischen Ausdruck als Funktionskörper. Um Konflikte mit anderen symbolischen Ausdrücken zu vermeiden, könnten man das `gensym`-Konstrukt (↪ Abschnitt 2.4.4 S. 259) verwenden. Wir nutzen hier einfach das Symbol `*F-NULL*` selbst.

```
eval> (define *F-NULL*
        (lambda ()
          *F-NULL*))
```

Exkurs: *Funktionale Konstante*
Die Funktion `(lambda (x) *F-NULL*)` wäre kein zweckmäßiges Beispiel für einen konstanten Rückgabewert; denn ihre Applikation kann unter Umständen zum Fehler führen:

```
eval> ((lambda (X) *F-NULL*)
         (/ 1 0)) ==> ERROR ...
                      ;Divison durch Null
```

Die Aufgabe eines Konstruktors ist das Zusammenfügen von Teilen. Beim `cons`-Konstrukt wird der `car`-Teil mit dem `cdr`-Teil verknüpft. Diese Verknüpfung ist abbildbar, indem wir die beiden Teile als Werte im Namensraum (Umgebung) einer Funktion speichern. Dazu bedienen wir uns des Konstruktes `define-syntax-rule`, das wir bei der Thematik *Makros* näher behandeln (↪ Abschnitt 2.5.3 S. 293).

```
;;;F-Cons
eval> (define *F-NULL*
        (lambda() *F-NUll*))
eval> (define-syntax-rule (f x y)
         (lambda ()
           (let ((ns (make-base-namespace)))
             (eval '(define x '*F-NULL*) ns)
             (eval '(define y '*F-NULL*) ns)
             ns)))

;;F-cons als Funktion deren Applikation
;; ein Namensraum ist.
eval> (define F-cons ;Anstelle von x und y
                     ; wird car-Teil und
                     ; cdr-Teil gesetzt.
        (f car-Teil cdr-Teil))
```

Die Expansion (Ergebnis) des obigen `define-syntax-rule`-Konstruktes ist:

```
eval> (define F-cons
        (lambda ()
          (let ((ns (make-base-namespace)))
            (eval '(define car-Teil '*F-NULL*) ns)
            (eval '(define cdr-Teil '*F-NULL*) ns)
            ns))))
```

Damit wird deutlich, dass `car-Teil` und `cdr-Teil` nur im Namensraum `ns` gebunden sind:

```
eval> (F-cons) ==> #<namespace:0>
eval> (eval 'car-Teil (F-cons))
      ==> *F-NULL*
eval> (eval 'car-Teil)
      ==> ERROR ...
eval> (define F-set-cdr!
        (lambda ()
          (let ((ns (F-cons)))
            (eval
             '(set! cdr-Teil
                    '(lambda(x) x))
             ns)
    ns)))
eval> (eval 'cdr-Teil (F-set-cdr!))
      ==> (lambda (x) x)
eval> (eval 'car-Teil (F-set-cdr!))
      ==> *F-NULL*
eval> (eval 'cdr-Teil (F-cons))
      ==> *F-NULL*
```

Statt den Namensraum (die Umgebung) explizit als Rückgabewert zu definieren, verändern wir das `F-cons`-Konstrukt so, dass sein Wert eine LISP-Funktion ist.

```
eval> (define F-cons
        (lambda (car_Teil cdr_Teil)
          (lambda (Operation)
            (cond ((eq? 'car Operation) car_Teil)
                  ((eq? 'cdr Operation) cdr_Teil)
                  (#t #f)))))
eval> (F-cons 'A 'B) ==> #<procedure>
eval> ((F-cons 'A 'B) 'car) ==> A
```

Diese (Rückgabe-)Funktion wurde in einer Umgebung erzeugt, in der `car_Teil` an den Wert `A` und `cdr_Teil` an den Wert `B` gebunden sind. Erzeugt heißt: Aus der Liste mit dem Symbol `lambda` in Präfixnotation ist durch Evaluieren eine LISP-Funktion, nach außen durch `#<procedure>` angezeigt, entstanden. Wird diese (Rückgabe-)Funktion angewendet, dann haben ihre freien Variablen, hier `car_Teil` und `cdr_Teil` die Werte der Definitionsumgebung, weil in Scheme als Standardfall die lexikalische Bindungsstrategie implementiert ist. Sie legt fest, dass die Definitionsumgebung einer LISP-Funktion maßgeblich für die Bindung von freien Variablen ist (↪ S. 42). Daher definieren wir den Konstruktor `F-cons` wie folgt:

```
eval> (define F-cons
        (lambda (car_Teil cdr_Teil)
          (lambda(Operation)
            (cond
              ((eq? Operation 'F-pair?)
               (lambda () #t))
              ((eq? Operation 'F-car)
               (lambda() car_Teil))
              ((eq? Operation 'F-cdr)
               (lambda() cdr_Teil))
              ((eq? Operation 'F-set-car!)
               (lambda(x) (set! car_Teil x)))
              ((eq? Operation 'F-set-cdr!)
               (lambda (x) (set! cdr_Teil x)))
              (#t (error
                    "Unbekannte Operation: "
                    Operation))))))
```

Die Selektoren `car` und `cdr` erfordern eine Liste bzw. ein Punkt-Paar als Argument. Diese Voraussetzung ist mit dem Prädikat `pair?` prüfbar. Für die Selektoren `F-car` und `F-cdr` definieren wir ein entsprechendes Prädikat `F-pair?`.

```
eval> (define F-pair?
        (lambda (Objekt)
          (cond
```

```
            ((and (procedure? Objekt)
                  ((Objekt 'F-pair?))) #t)
            (#t (error
                 "Keine funktionale cons-Zelle: "
                 Objekt)))))
```

In klassischen LISP-Implementationen können die Selektoren `car` und `cdr` auf die leere Liste angewendet werden, obwohl ihre Werte dann prinzipiell nicht definiert sind. Der Rückgabewert ist dann oft `null`. In modernen Scheme-Implementationen wie z. B. *PLT-Scheme* (DrScheme) wird auf Fehler erkannt.

```
eval> (null? (cdr (list))) ==> ERROR ...
      cdr: expects argument
      of type <pair>; given ()
```

Wir definieren ein entsprechendes `F-null?`-Konstrukt. Ist die „leere Liste“ gegeben, hat unser `F-null?` den Wert `*F-NULL*`. Dieser Wert ist ungleich `false` und wird daher als „wahr“ interpretiert. Liegt nicht `*F-NULL*` vor, dann ist der Wert von `F-null?` gleich `false`. `*F-NULL*` übernimmt nicht die Mehrfachfunktion von `NIL` (Wahrheitswert-Repräsentation und „*bottom-element*“ einer Liste), wie in klassischen LISP-Systemen üblich. `*F-NULL*` dient nur als „Urbaustein“ für den funktionalen Konstruktor `F-cons`.

```
eval> (define F-null?
        (lambda (Objekt)
          (cond
            ((and (procedure? Objekt)
                  (eq? Objekt *F-NULL*))
             *F-NULL*)
            (#t #f))))
eval> (define F-car
        (lambda (Objekt)
          (cond
            ((F-null? Objekt) Objekt)
            ((F-pair? Objekt)
              ((Objekt 'F-car)))
            (#t #f))))
eval> (define F-cdr
        (lambda (Objekt)
          (cond
            ((F-null? Objekt) Objekt)
            ((F-pair? Objekt)
              ((Objekt 'F-cdr)))
            (#t #f))))
eval> (define F-set-car!
        (lambda (Objekt Wert)
          (cond
            ((F-null? Objekt)
```

```
             (error "Objekt ist leer: "
                    Objekt))
            ((F-pair? Objekt)
             ((Objekt 'F-set-car!)
               Wert))
            (#t #f))))
eval>(define F-set-cdr!
        (lambda (Objekt Wert)
          (cond
            ((F-null? Objekt)
             (error "Objekt ist leer: "
                    Objekt))
            ((F-pair? Objekt)
             ((Objekt 'F-set-cdr!)
               Wert))
            (#t #f))))
```

Das `F-cons`-Beispiel zeigt den (gewollten) Nebeneffekt des Mutators `F-set-car!`. Seine Wirkung entspricht der von `set-car!`.

```
eval> (define Foo (F-cons 'A *F-NULL*))
eval> (define Foo (F-cons 'B Foo))
eval> (define Baz Foo)
eval> (F-car Foo) ==> B
eval> (F-car Baz) ==> B
eval> (F-cdr Baz) ==> #<procedure>
eval> (F-car (F-cdr (F-cdr Baz)))
      ==> #<procedure:*F-NULL*>
eval> (F-set-car! Baz (F-cons 1 *F-NULL*))
eval> (F-car Foo) ==> #<procedure>
eval> (F-car (F-car Foo)) ==> 1
eval> (F-car (F-cdr Foo)) ==> A
eval> (F-car (F-cdr (F-cdr Foo)))
       ==> #<procedure:*F-NULL*>
```

Ausgangspunkt einer (noch stärker) funktionsgeprägten Abbildung der cons-Zelle können Funktionen für die Wahrheitswert `true` und `false` sein.

```
eval> (define True
        (lambda (W F) W))
eval> (define False
        (lambda (W F) F))
```

Das `F-cons`-Konstrukt ist wieder eine Funktion höherer Ordnung. Ihr Rückgabewert ist wieder eine Funktion, in deren Funktionskörper die `car`- und `cdr`-Werte als Argumente dienen.

```
eval> (define F-cons
        (lambda (x y)
          (lambda (c)
            (c x y))))
```

Mit Hilfe der `True`- und `False`-Funktionen können die Selektoren als Funktionen, wie folgt, definiert werden:

```
eval> (define F-car
        (lambda (x)
          (x True)))
eval> (define F-cdr
        (lambda (x)
          (x False)))
```

Der Wert einer `F-cons`-Applikation ist eine Funktion, die wiederum von den Selektoren appliziert wird.

```
eval> (F-cdr (F-cons 2 (F-cons 1 0)))
      ==> #<procedure>
eval> (F-car (F-cons 2 (F-cons 1 0)))
      ==> 2
eval> (F-cdr (F-cdr (F-cons 1 0)))
      ==> ERROR ...
          ;procedure application:
          ; expected procedure,
          ; given: 0; arguments were:
          ; #<procedure:False>
eval> (F-cdr (F-cons 1 0)) ==> 0
```

Zum besseren Verstehen eines Selektionsvorganges, „reduzieren“ wir schrittweise die `lambda`-Konstrukte, d. h. wir setzen die Werte ihrer Argumente im Funktionskörper an die Stelle der entsprechenden `lambda`-Variablen. Die einzelnen Reduktionsschritte trennen wir mit dem Zeichen ":=".

```
eval> (F-cdr (F-cons 1 0))
:=    (F-cdr ((lambda (x y) (lambda (c) (c x y)))
                                              1 0))
:=    (F-cdr (lambda (c) (c 1 0)))
:=    ((lambda (x) (x False)) (lambda (c) (c 1 0)))
:=    ((lambda (c) (c 1 0)) False)
:=    (False 1 0)
:=    ((lambda (W F) F) 1 0)    ==> 0
```

Fixpunktoperator `Y`

Wir betrachten jetzt Funktionen, die sich selbst als Argument haben. Der sogenannte Fixpunktoperator *Y* ist eine solche Funktion. Zu seiner Erläuterung nehmen wir als Beispiel an, dass die Summe der numerischen Elemente einer Liste zu berechnen ist. Die Beispielliste nennen wir `Zahlungen`.

```
eval> (define Zahlungen
        '(10.00 "Dubios" 5.00 55.00
          "Dubios" 2.00 8.00))
```

Mit dem Iterationskonstrukt `for-each` und einer Hilfsvariablen `Akkumulator` ist die Summe einfach ermittelbar:

```
eval> (let ((Akkumulator 0))
       (for-each
        (lambda (x)
         (if (number? x)
             (set! Akkumulator
                 (+ Akkumulator x))
             x))
        Zahlungen)
      Akkumulator) ==> 80.0
```

An Stelle des iterativen `for-each`-Konstruktes mit dem eingebetteten Modifikator `set!` („Zuweisungsoperator“) definieren wir eine rekursive Funktion.

```
eval> (define Summe
        (lambda (Liste)
          (cond ((null? Liste) 0)
                ((number? (car Liste))
                 (+ (car Liste)
                    (Summe (cdr Liste))))
                (#t (Summe (cdr Liste))))))
eval> (Summe Zahlungen) ==> 80.0
```

Zur Vermeidung einer globalen Funktion ist `Summe` jetzt als lokale Funktion innerhalb eines `letrec`-Konstruktes definiert.

```
eval> (letrec
          ((Summe (lambda (Liste)
                    (cond ((null? Liste) 0)
                          ((number? (car Liste))
                           (+ (car Liste)
                              (Summe
                               (cdr Liste))))
                          (#t (Summe
                               (cdr Liste)))))))
        (Summe Zahlungen)) ==> 80.0
```

Unser Ziel ist eine rekursive Lösung ohne die Nutzung des `letrec`-Konstruktes. Gesucht ist ein Mechanismus, der rekursive Lösungen konstruieren kann, die nur auf dem `lambda`-Konstrukt bzw. dem `let`--Konstrukt (als dessen syntaktische Vereinfachung) basieren und kein `set!`-Konstrukt verwenden. Hier wollen wir daher nicht das `letrec`--Konstrukt entsprechend Abschnitt 1.2.2 S. 69 diskutieren, da dieses auf dem `set!`-Konstrukt aufbaut.

Ein solcher Mechanismus ist der Fixpunktoperator *Y* (engl.: *the applicative-order fixed point operator for functionals*). Der Fixpunktoperator *Y*

erfüllt die folgende Bedingung:

$$((F\ (Y\ F))\ x) = ((Y\ F)\ x)$$

Anders formuliert:

$$((F\ f_p)\ x) = (f_p\ x)$$

Dabei ist f_p der Fixpunkt von F, d. h. F „überführt“ f_p in f_p. Für unser Problem ist das `F`-Konstrukt, wie nach den weiteren Ausführungen deutlich wird, folgendermaßen definiert:

```
eval> (define F
        (lambda (Summe)
          (lambda (Liste)
            (cond ((null? Liste) 0)
                  ((number? (car Liste))
                   (+ (car Liste)
                      (Summe (cdr Liste))))
                  (#t (Summe (cdr Liste)))))))
```

Der Fixpunktoperator Y ist eine Funktion, die eine Funktion nimmt, die als rekursive Funktion betrachtbar ist und eine andere Funktion zurückgibt, welche die rekursive Funktion implementiert. Die Befassung mit dem Fixpunktoperator Y zwingt zum funktionalen Denken, wie der vorherstehende Satz offensichtlich demonstriert.

Zum Verstehen des Fixpunktoperators entwicklen wir schrittweise eine Y-Definition (↪ [68]). Dazu nutzen wir die drei Überlegungen:

1. Einführung einer zusätzlichen `lambda`-Variablen zur Vermeidung von rekursiven Konstrukten, wie `letrec` oder `define`.

2. Konvertierung einer Funktion mit mehreren `lambda`-Variablen in geschachtelte Funktionen mit einer `lambda`-Variablen (*Curryfizierung*). Ziel ist, die Manipulation der (neuen) `lambda`-Variablen für den Selbstbezug von der Manipulation der bisherigen `lambda`-Variablen zu trennen.

3. Definition gleichwertiger Funktionen durch Analyse eines Konstruktes (*Abstraktion*).

Zum (Wieder-)Einfinden in die funktionale Denkwelt betrachten wir zunächst die folgende einfache Funktion `Foo`. Sie hat eine Funktion als Rückgabewert.

```
eval> (define Foo
        (lambda (n)
          (lambda (m)
            (display "Wert von n = ")
            (display n)
```

```
            (newline)
            (display "Wert von m = ")
            (display m)
            (newline)
            (* n m))))
eval> ((Foo 2) 3) ==>
Wert von n = 2
Wert von m = 3
6
```

Wenden wir uns nun der Definition der Funktion `Summe` zu und nennen sie für die folgende Erörterung vorübergehend `<summe>`.

```
    (letrec
        ((<summe> (lambda (Liste)
                    (cond ((null? Liste) 0)
                          ((number? (car Liste))
                           (+ (car Liste)
                              (Summe
                               (cdr Liste))))
                          (#t (Summe
                               (cdr Liste)))))))
        (<summe> Zahlungen))
```

Im Sinne der rekursiven Problemlösungsmethode, bei der angenommen wird, dass eine Teillösung (quasi) existiert, unterstellen wir auch hier, die Funktion `Summe` wäre verfügbar (woher auch immer). Wenn die Funktion `Summe` schon (gedanklich) existiert, dann könnte sie allerdings auch über die lambda-Schnittstelle dem Funktionskörper von `<summe>` übergeben werden. Die Funktion `<summe>` hätte dann zwei `lambda`-Variablen: `Funktion` für die zu übergebende Funktion `Summe` und `Liste`, die bisherige Variable. Wenn wir im Funktionskörper die übergebene Funktion aufrufen, also `Summe`, dann ist zu beachten, dass diese jetzt zwei Argumente benötigt.

```
eval> (let ((Summe
             (lambda (Funktion Liste)
               (cond ((null? Liste) 0)
                     ((number? (car Liste))
                      (+ (car Liste)
                         (Funktion Funktion
                                   (cdr Liste))))
                     (#t (Funktion Funktion
                                   (cdr Liste)))))))
        (Summe Summe Zahlungen)) ==> 80.0
```

Zunächst bindet das `let`-Konstrukt in seiner Umgebung das Symbol `Summe` an eine Funktion mit den beiden `lambda`-Variablen `Funktion` und `Liste`. Diese Funktion hat keinen Selbstbezug zum Symbol `Summe`. In ihrem Funktionskörper wird die `lambda`-Variable `Funktion` einerseits appliziert und andererseits als Argument genutzt. Die Besonderheit

ist das Zusammenfallen der Punkte „applizierte Funktion“ und „Argument“. Wird die `lambda`-Variable `Funktion` durch die Notation als erstes Element einer Liste als Funktion appliziert, dann ist sie selbst ihr erstes Argument. Die Technik, sich selbst als Argument zu nutzen, ergibt den rekursiven Bezug. Etwas leger formuliert: Wir „überlisten“ das `let`-Konstrukt, indem wir die Rekursivität „durch die Hintertür“ als Parameterübergabe realisieren.

Die zusätzliche `lambda`-Variable für die Selbstübergabe wollen wir von den anderen `lambda`-Variablen trennen, mit dem Ziel, den Selbstübergabe-Mechanismus isoliert betrachten zu können. Für diese Trennung nutzen wir das sogenannte *Curryfizieren* von Funktionen.

H. B. Curry[14] hat nachgewiesen, dass zu jeder Funktion höherer Ordnung F eine Funktion F^c existiert, für die gilt ($\hookrightarrow$ [44]):

$$(F\ x_1\ x_2\ x_3 \ldots x_n) = (\ldots(((F^c\ x_1)\ x_2)\ x_3)\ldots x_n)$$

Das folgende Beispiel zeigt das Curryfizieren (engl.: *currying*) anhand von drei `lambda`-Variablen:

```
eval> (define Foo
        (lambda (x y z)
          (list x y z)))
eval> (Foo 'a 'b 'c)
      ==> (a b c)
eval> (define Curryfizierte-Foo
        (lambda (x)
          (lambda (y)
            (lambda (z)
              (list x y z)))))
eval> (((Curryfizierte-Foo 'a) 'b) 'c)
      ==> (a b c)
```

Bezogen auf die Definition von `Summe` erhalten wir:

```
eval> (let ((Summe
             (lambda (Funktion)
               (lambda (Liste)
                 (cond ((null? Liste) 0)
                       ((number? (car Liste))
                        (+ (car Liste)
                           ((Funktion Funktion)
                              (cdr Liste))))
                       (#t ((Funktion Funktion)
                              (cdr Liste)))))))))
            ((Summe Summe) Zahlungen)) ==> 80.0
```

[14] *Haskell Brooks Curry*, $\star$ 12-Sep-1990 in Millis, Massachusetts — † 1-Sep-1982 in State College, Pennsylvania, war ein amerikanischer Mathematiker.

Das Curryfizieren hat eine Hierarchie von zwei `lambda`-Konstrukten bewirkt und die `lambda`-Variable `Funktion` auf die obere Ebene gehoben. Die Selbstübergabe `(Funktion Funktion)` ist jedoch noch tief in dem Konstrukt `(lambda (Liste)...)` enthalten. Der nächste Schritt gilt dem Herauslösen von `(Funktion Funktion)`. Dazu definieren wir eine lokale Funktion `Sum` mit den beiden `lambda`-Variablen: `Fkt` und `Lst`. Wir übergeben `Sum` unseren Selbstbezug `(Funktion Funktion)` als erstes Argument.

```
eval> (let ((Summe
        (lambda (Funktion)
         (lambda (Liste)
           (let ((Sum
                  (lambda (Fkt Lst)
                    (cond ((null? Lst) 0)
                          ((number? (car Lst))
                           (+ (car Lst)
                              (Fkt (cdr Lst))))
                          (#t (Fkt (cdr Lst)))))))
             (Sum (Funktion Funktion) Liste))))))
    ((Summe Summe) Zahlungen))  ==> 80.0
```

Da die lokale Funktion `Sum` zwei `lambda`-Variablen: `Fkt` und `Lst` hat, Curryfizieren wir `Sum`. Dabei ist zu beachten, dass auch der `Sum`-Aufruf anzupassen ist.

```
eval> (let ((Summe
        (lambda (Funktion)
         (lambda (Liste)
           (let ((Sum
                  (lambda (Fkt)
                    (lambda (Lst)
                    (cond ((null? Lst) 0)
                          ((number? (car Lst))
                           (+ (car Lst)
                              (Fkt (cdr Lst))))
                          (#t (Fkt (cdr Lst))))))))
             ((Sum (Funktion Funktion)) Liste))))))
    ((Summe Summe) Zahlungen)) ==> 80.0
```

Analysiert man die bisher erreichte Lösung, so zeigt sich, dass es keinen Grund gibt, die lokale Funktion `Sum` auf einer so tiefen Ebene zu definieren. Wir können sie z. B. als eine globale Funktion `Sum` definieren, so dass sie der obigen `F`-Definition entspricht. Unsere Lösung besteht dann aus der `Sum`-Definition und dem Rest des `let`-Konstruktes:

```
eval> (define Sum (lambda (Fkt)
                (lambda (Lst)
                    (cond ((null? Lst) 0)
                          ((number? (car Lst))
                           (+ (car Lst)
```

```
                              (Fkt (cdr Lst))))
                            (#t (Fkt (cdr Lst)))))))
eval> (let ((Summe (lambda (Funktion)
         (lambda (Liste)
           ((Sum (Funktion Funktion)) Liste)))))
        ((Summe Summe) Zahlungen)) ==> 80.0
```

Die Analyse dieser Lösung ergibt: Für eine allgemeine Anwendung ist es ungünstig, dass in der lokalen Funktion `Summe` die Funktion `Sum` explizit aufgerufen wird. Um diesen Mangel zu beheben, bauen wir die Definition von `Summe` in eine Funktion `Y0` ein, deren `lambda`-Variable `Fun` dann an `Sum` gebunden wird, so dass `Fun` appliziert werden kann. Der explizite `Sum`-Aufruf wird durch einen `Fun`-Aufruf ersetzt. Bei der `Y0`-Definition sorgen wir dafür, dass ihr Rückgabewert `(Summe Summe)` ist. Die Lösung hat dann folgende Struktur:

```
(let ((S (Y0 Sum)))
     (S Zahlungen))
```

und `Y0` folgende Definition:

```
eval> (define Y0
        (lambda (Fun)
          (let ((Summe
                 (lambda (Funktion)
                   (lambda (Liste)
                     ((Fun
                       (Funktion Funktion))
                      Liste)))))
            (Summe Summe))))
```

Die so abstrahierte Definition `Y0` stellt den Fixpunktoperator `Y` dar. Für seine allgemeine Anwendung benennen wir die `lambda`-Variablen von `Y0` um, und zwar: `Fun` in `F`, `Summe` in `G`, `Funktion` in `H` und `Liste` in X. Damit erhalten wir das folgende `Y`-Konstrukt:

```
eval> (define Y
        (lambda (F)
          (let ((G (lambda (H)
                     (lambda (X)
                       ((F (H H)) X)))))
            (G G))))
```

Das Erläuterungsbeispiel, die Summe der numerischen Elemente einer Liste, ergibt mit dem Fixpunktoperator `Y` und der eingesetzten `Sum`-Definition damit folgende Lösung:

```
eval> (let ((S (Y
            (lambda (Fkt)
              (lambda (Lst)
                (cond ((null? Lst) 0)
```

```
                      ((number? (car Lst))
                       (+ (car Lst)
                          (Fkt (cdr Lst))))
                      (#t (Fkt (cdr Lst)))))))))
    (S Zahlungen)) ==> 80.0
```

Diese Y-Definition hat die oben geforderte Eigenschaft eines Fixpunktoperators. Das Y-Konstrukt erfüllt die folgende Gleichung (F ↪ S. 279):

```
eval> (equal? ((F (Y F)) Zahlungen)
              ((Y F) Zahlungen))
      ==> #t
```

Der bisherige Y-Operator ist definiert, um eine rekursive Funktion einzurichten. Damit eine Erweiterung auf zwei gegenseitig rekursive Funktionen möglich wird, ändern wir die Y-Definition, wie folgt:

```
eval> (define Y
        (lambda (F)
          (let ((G (lambda (H R)
                     (lambda (X)
                       ((R (H H F)) X)))))
            (G G F))))
```

Es wurde eine zusätzliche lambda-Variable R eingeführt, damit F stets als Argument verwendet wird. Diese Y-Definition hat den selben Effekt. Allerdings hat G eine zusätzliche lambda-Variable. So ist es möglich, dass G mit mehr als einer Funktion auf einmal umgehen kann. Zum leichteren Verstehen benennen wir in Y die Funktionen um und zwar H in G:

```
eval> (define Y
        (lambda (F)
          (let ((G (lambda (G R)
                     (lambda (X)
                       ((R (G G F)) X)))))
            (G G F))))
eval> (define Y-2
        (lambda (F H)
          (let ((G (lambda (G R)
                     (lambda (N)
                       ((R (G G F) (G G H))
                        N)))))
        (G G F))))
```

Ein einfaches Beispiel für die Y-2-Nutzung ist die Prüfung, ob eine vorgegebene Zahl gerade oder ungerade ist (↪ [68]).

```
eval> ((Y-2
        (lambda (F H)
          (lambda (N) (if (< N 1)
```

```
                    "Gerade" (H (- N 1)))))
        (lambda (F H)
          (lambda (N) (if (< N 1)
                          "Ungerade" (F (- N 1))))))
       13) ==> "Ungerade"
```

Auffällig ist die große Ähnlichkeit zum entsprechenden `letrec`-Konstrukt:

```
eval> (letrec
          ((F (lambda (N) (if (< N 1)
                              "Gerade" (H (- N 1)))))
           (H (lambda (N) (if (< N 1)
                              "Ungerade" (F (- N 1))))))
        (F 13)) ==> "Ungerade"
```

2.5.2 Kontrollstruktur: *Continuation*

Kennzeichnend für funktionsgeprägte Konstruktionen ist die Komposition von Funktionen (≡ evaluierte `lambda`-Konstrukte). Die *EVAL*-Regeln legen ihre Abarbeitungs-Reihenfolge („Kontrollstruktur“ ↪ Abschnitt 1.2.2 S. 65) fest, d. h. sie bestimmen, welche Funktion wann auszuwerten ist. Kern der Kontrollstruktur ist die Sequenz:

1. Bestimmung der Werte der Argumente und
2. Ermittlung des Funktionswertes mit den übernommenen Werten der Argumente (↪ *EVAL*-Regel 2; ↪ S. 22).

Beispielsweise hat die Komposition `(F (G (H Foo)))` die folgende Kontrollstruktur (mit $S_i \equiv \underline{S}\text{chritt}_i$):

S_1 Wert von `F` ermitteln.
Da dieser Wert eine Funktion ist, ist zunächst sein Argument `(G (H Foo))` zu evaluieren, daher:

S_2 Wert von `G` ermitteln.
Da dieser Wert eine Funktion ist, ist zunächst sein Argument `(H Foo)` zu evaluieren, daher:

S_3 Wert von `H` ermitteln.
Da dieser Wert eine Funktion ist, ist zunächst sein Argument `Foo` zu evaluieren, daher:

S_4 Wert von `Foo` ermitteln. Dieser sei `v`.

S_5 `v` in `H` übernehmen und Funktionswert `H` berechnen. Dieser Wert sei `w`.

S_6 `w` in `G` übernehmen und Funktionswert `G` berechnen. Dieser Wert sei `x`.

S_7 `x` in `F` übernehmen und Funktionswert `F` berechnen. Dieser Wert sei `y`.

S_8 `y` ist der Wert der Komposition.

Bei jedem Schritt S_i der Folge S_1 bis S_8 steht fest, welcher Schritt S_{i+1} ist, d. h. die Fortsetzung (engl.: *continuation*) der Abarbeitung ist bekannt. Befinden wir uns beispielsweise beim Schritt S_5, dann besteht die zukünftige Abarbeitung in der Applikation der Funktionen `G` und `F`. Dieses „Kontrollwissen“ über die Zukunft hätte man gern beim Abarbeiten von S_5 direkt verfügbar. Man könnte dann z. B. entscheiden, ob es sich lohnt die „Zukunft“ überhaupt abzuarbeiten. Ist das „Kontrollwissen“ als Wert einer Variable speicherbar, dann kann man die Abarbeitung später fortsetzen. Außerdem ist sie dann beliebig oft wiederholbar. Scheme stellt mit dem Konstrukt `call-with-current-continuation`, kurz: `call/cc`, einen entsprechenden Mechanismus bereit. Als Variable, die an die „Zukunft“ gebunden ist, dient die `lambda`-Variable des `lambda`-Konstruktes, das mit `call/cc` appliziert wird. Die Funktionsweise des `call/cc`-Konstruktes erläutern wir anhand der obigen Komposition `(F (G (H Foo)))`. Darauf aufbauend wird gezeigt, dass das `call/cc`-Konstrukt als eine allgemeine Kontrollstruktur betrachtbar ist, die beliebige Sprünge in die „Zukunft“ und „Vergangenheit“ einer Abarbeitung ermöglicht. Mit dem `call/cc`-Konstrukt ist daher das klassische LISP-Sprung-Paar `CATCH` und `THROW` leicht abbildbar (↪ Abschnitt 2.5.4 S. 300).

Vorab definieren wir für unsere Kompostion `(F (G (H Foo)))` ein simples Beispiel, wie folgt:

```
eval> (define Foo 7)
eval> (define H
        (lambda (N)
          (display "Funktion H: ")
          (display N)
          (newline)
          (+ N 1)))
eval> (define G
        (lambda (N)
          (display "Funktion G: ")
          (display N)
          (newline)
          (+ N 1)))
eval> (define F
        (lambda (N)
          (display "Funktion F: ")
          (display N)
```

```
          (newline)
          (+ N 1)))
eval> (F (G (H Foo))) ==>
      Funktion H: 7
      Funktion G: 8
      Funktion F: 9
      10
```

Die „Zukunft“, gesehen vom Zeitpunkt der Abarbeitung von `H` aus, speichern wir als globale Variable `*KONTROLLWISSEN-BEI-H*`. Diese Variable erhält den Wert über das `call/cc`-Konstrukt mit Hilfe des `set!`-Konstruktes. Bei unserem `call/cc`-Konstrukt ist die „Zukunft“ an die `lambda`-Variable `Fortsetzung` gebunden, so dass wir sie beim Abarbeiten von `call/cc` an die globale Variable binden können.

```
eval> (define *KONTROLLWISSEN-BEI-H* null)
eval> (define H
        (lambda (N)
          (call/cc
            (lambda (Fortsetzung)
              (set! *KONTROLLWISSEN-BEI-H* Fortsetzung)
              (display "Funktion H: ")
              (display N)
              (newline)
              (+  N 1)))))
eval> (F (G (H Foo))) ==>
      Funktion H: 7
      Funktion G: 8
      Funktion F: 9
      10
eval> *KONTROLLWISSEN-BEI-H*
      ==> #<continuation> ;Der Datentyp hat
                          ; keine aussage-
                          ; fähige externe
                          ; Darstellung,
                          ; daher diese
                          ; Ersatzdarstellung
                          ; analog zur #<procedure>
```

Die gespeicherte `#<continuation>` kann nun erneut ausgeführt werden, d. h. wir setzten wieder bei `H` auf. Der Wert von `H`, also das Resultat der bisherigen Verarbeitung, ist als Argument angebbar.

```
eval> (*KONTROLLWISSEN-BEI-H* 8) ==>
      Funktion G: 8
      Funktion F: 9
      10
```

Die *Continuation* `*KONTROLLWISSEN-BEI-H*` kann mit einem anderen Wert als `8` appliziert werden, d. h. die ausstehende Berechnung ist von der Stelle `H` mit einem neuen Wert durchführbar. Wir springen quasi in die Berechnung mit einem neuen Startwert zurück.

```
eval> (*KONTROLLWISSEN-BEI-H* 13) ==>
      Funktion G: 13
      Funktion F: 14
      15
```

Der *Contiunation*-Mechanismus ermöglicht einerseits Sprünge aus einer Berechnung heraus, andererseits Sprünge in eine Berechnung hinein. Etwas salopp formuliert: Berechnungen sind jederzeit einfrierbar und jederzeit auftaubar. Legen wir eine Berechnung auf Eis, dann sind die schon vollzogenen Bindungen von Variablen miteinzufrieren. Erst die korrekte Behandlung dieser aktuellen Bindungen macht den *Continuation*-Mechanismus zu einer nützlichen Kontrollstruktur. Das einfache Heraus- und Hereinspringen in eine Folge von Anweisungen ermöglicht ein *GO-TO*-Konstrukt, das viele Programmiersprachen bereitstellen. Davon zu unterscheiden ist der Rücksprung mit einer Aktivierung einer eingefrorenen, vormals aktuellen Umgebung.

Wir verdeutlichen diese Eigenschaft des `call/cc`-Konstruktes anhand des folgenden Beispiels. Es prüft, ob in einer Liste von Zahlen eine negative Zahl vorkommt. Zur Iteration über die Liste nutzen wir das `map`-Konstrukt. Zur Vereinfachung der Ausgabe definieren wir `DisplayLn` für *Display Line*.

```
eval > (define DisplayLn (lambda (x)
                           (display x)
                           (newline)))
eval> (map (lambda (x) (if (>= 0 x)
                       (begin (DisplayLn x) #t)
                        #f))
       '(1 2 -3 4 -5)) ==>
      -3
      -5
      (#f #f #t #f #t)
```

Dieses `map`-Konstrukt notieren wir in geschachtelte `lambda`-Konstrukte, um Bindungswert innerhalb und außerhalb des `map`-Konstruktes zeigen zu können.

```
eval> (define Foo
        (lambda (n)
          (let ((Var n))
            (set! Var (* 2 n))
            (lambda (L)
              (map (lambda (x)
                (if (>= 0 x)
                    (begin (DisplayLn x) #t)
                    #f))
               L)
            (DisplayLn "Map Vollzug")
            (DisplayLn
```

```
            "2.Schritt mit Bindungen ")
          (display "Aussen: ")
          (display Var)
          (display " Innen: ")
          (DisplayLn L)
          (DisplayLn "n.Schritt")))))
eval> ((Foo 6) '(1 2 -3 4 -5)) ==>
      -3
      -5
      Map Vollzug
      2.Schritt mit Bindungen
      Aussen: 12 Innen: (1 2 -3 4 -5)
      n.Schritt
eval> ((Foo 6) '(1 2 3 4 5)) ==>
      Map Vollzug
      2.Schritt mit Bindungen
      Aussen: 12 Innen: (1 2 3 4 5)
      n.Schritt
```

Bei der ersten negativen Zahl sei die Analyse der Zahlenfolge zu beenden. Ist `x` negativ, dann arbeiten wir im `map`-Konstrukt die Fortsetzung des `map`-Konstruktes ab. Damit wir später von dieser Situation erneut die Abarbeitung starten können, wird sie an die globale Variable `*KONSTRUKTIONS-REST*` gebunden. Dazu applizieren wir das `map`-Konstrukt nicht direkt, sondern über das `call/cc`-Konstrukt.

```
eval> (define *KONSTRUKTIONS-REST* null)
eval> (define Foo
        (lambda (n)
          (let ((Var n))
            (set! Var (* 2 n))
            (lambda (L)
              (call/cc
               (lambda (Fortsetzung)
              (map (lambda (x)
               (if (>= 0 x)
                   (begin (DisplayLn x)
                     (set! *KONSTRUKTIONS-REST*
                           Fortsetzung)
                      ;;Dummy weil die
                      ;; Schnittstelle
                      ;; ein Argument
                      ;; erfordert
                      (Fortsetzung 'Dummy))
                   #f))
                L)))
          (DisplayLn "Vollzug")
          (DisplayLn
            "2.Schritt mit Bindungen ")
          (display "Aussen: ")
```

```
          (display Var)
          (display " Innen: ")
          (DisplayLn L)
          (DisplayLn "n.Schritt")))))
eval> ((Foo 6) '(1 2 -3 4 -5)) ==>
      -3
      Vollzug
      2.Schritt mit Bindungen
      Aussen: 12 Innen: (1 2 -3 4 -5)
      n.Schritt
```

An die `lambda`-Variable `Fortsetzung` ist die ausstehende Abarbeitung der `display`-Konstrukte einschließlich ihrer zugehörenden Umgebung gebunden, so dass `Var` und `L` ihre bisher aktuellen Werte haben. Da diese „Zukunft“ als `*KONSTRUKTIONS-REST*` gespeichert ist, können wir sie erneut ausführen. Die Ausgabe von `Var` und `L` zeigt wieder diese Werte, selbst wenn `*KONSTRUKTIONS-REST*` in einer entsprechend geänderten Umgebung ausgeführt wird.

```
eval> (continuation? *KONSTRUKTIONS-REST*)
      ==> #t
eval> (*KONSTRUKTIONS-REST* 'Dummy) ==>
      Vollzug
      2.Schritt mit Bindungen
      Aussen: 12 Innen: (1 2 -3 4 -5)
      n.Schritt
eval> (let ((Var 7)
            (L '(a b c d)))
        (*KONSTRUKTIONS-REST* 'Dummy)) ==>
      Vollzug
      2.Schritt mit Bindungen
      Aussen: 12 Innen: (1 2 -3 4 -5)
      n.Schritt
```

Enthält die Folge keine negativen Zahlen, dann entspricht die `Foo`-Definition mit dem `call/cc`-Konstrukt der ursprünglichen Fassung.

```
eval> (define *KONSTRUKTIONS-REST* null)
eval> ((Foo 2) '(1 2 3)) ==>
      Vollzug
      2.Schritt mit Bindungen
      Aussen: 4 Innen: (1 2 3)
      n.Schritt
eval> (continuation? *KONSTRUKTIONS-REST*)
      ==> #f
```

Der *Continuation*-Mechanismus erzeugt „Einheiten“ im Sinne von *first-class*-Objekten (↪ Abschnitt 1.2.2 S. 65). Sie sind daher auch als Argument einsetzbar. Wir können eine Continuation mit sich selbst als ein Argument aufrufen. Einen solchen Fall zeigt das folgende Konstrukt

`Summe-m->n`. Es ermittelt die Summe der Integerzahlen von der unteren Grenze `m` bis zur oberen Grenze `n`. Zum Verständnis dieser Selbstübergabe bei *Continuations* diskutieren wir vorab folgendes einfache Beispiel:

```
eval> (define Foo
        (list
         (call/cc (lambda (-c-) #t))))
eval> Foo ==> (#t)
```

Beim Auswerten dieses `call/cc`-Konstruktes sind an die `lambda`-Variable `-c-` die zu diesem Zeitpunkt noch ausstehenden Abarbeitungsschritte gebunden. Es sind die Schritte: Ausführung von `list` und Bindung von `Foo` an den Wert von `list`. Nach dem Auswerten hat das `call/cc`-Konstrukt den Wert `#t`. Die Bindung von `-c-` ist wieder aufgelöst. Sie wurde hier nicht genutzt. Wir haben eine Situation, als wäre folgende Formulierung auszuwerten:

```
eval> (define Foo (list #t))
```

In der folgenden Formulierung nutzen wir die Bindung von `-c-`.

```
eval> (define Foo
        (list
         (call/cc (lambda (-c-) -c-))))
```

Der Rückgabewert des `call/cc`-Konstruktes ist der Wert von `-c-`, also die noch ausstehenden Abarbeitungsschritte in Form einer Continuation. Mit dieser *Continuation* bildet `list` eine Liste und `define` bindet `Foo` an diese Liste.

```
eval> > Foo ==> (#<continuation>)
```

Wir greifen auf diese *Continuation* zurück und veranlassen die Ausführung der zunächst nicht vollzogenen Abarbeitungsschritte.

```
eval> (car Foo) ==> #<continuation>
eval> ((car Foo) #t)
eval> Foo ==> (#t)
```

Es sei die Summe einer Integerzahlenfolge mit den Grenzwerten `m` und `n` zu bestimmen. Die Lösung bedingt die Abbildung einer Wiederholung. Eine solche Wiederholung ist als Rekursion realisierbar. Die Rekursion können wir simulieren, indem wir einer Funktion sich selbst als Argument übergeben. Diese Technik haben wir beim Fixpunktoperator *Y* genutzt (↪ Abschnitt 2.5.1 S. 277). Jetzt übergeben wir einer *Continuation* sich selbst als Argument. Anders formuliert: Die `Continuation` repräsentiert die Zukunft, d. h. die (noch) abzuwickelnden Schritte. Ihr Argument repräsentiert den Startwert dafür, d. h. den Wert der vollzogenen Schritte („Vergangenheit“).

```
eval> (define Summe-m->n
    (lambda (m n)
      ;;Fallunterscheidung 1
      (cond ((< m n)
             (let* ((Summe 0)
                    (Verarbeitung
                     (call/cc (lambda (-c-) -c-))))
               ;;Fallunterscheidung 2
               (cond ((= m n )
                      (+ m Summe))
                     (#t (set! Summe (+ n Summe))
                         (set! n (- n 1))
                         (display Verarbeitung)
                         (display " n: ")
                         (display n)
                         (display " Summe: ")
                         (display Summe)
                         (newline)
                         (Verarbeitung Verarbeitung)))))
            (#t (display "Von-Wert: ")
                (display m)
                (display " nicht grösser bis-Wert: ")
                (display n)))))
eval> (Summe-m->n 4 2) ==>
      Von-Wert: 4 nicht grösser bis-Wert: 2
eval> (Summe-m->n 2 5) ==>
      #<continuation> n: 4 Summe: 5
      #<continuation> n: 3 Summe: 9
      #<continuation> n: 2 Summe: 12
      14
```

Zum Verstehen des Konstruktes `Summe-m->n` ist die Bindung der `lambda`-Variablen `-c-` bedeutsam. Ihr Wert repräsentiert die ausstehenden Abarbeitungsschritte, die das `cond`-Konstrukt (↪ Fallunterscheidungen 2) definiert. Aufgrund des `let*`-Konstruktes gehört die Bindung der lokalen Variablen `Summe` nicht zu den offenen Arbeitsschritten zum Zeitpunkt der Bindung von `-c-`. Definiert man `Summe-m->n` allerdings mit `let`, dann umfasst die *Continuation* `-c-` die Bindung von `Summe` an den Wert `0`. Man erhält damit das folgende, falsche Ergebnis:

```
eval> ;;Mit let statt mit let* definiert
     (Summe-m->n 2 5) ==>
     #<continuation> n: 4 Summe: 5
     #<continuation> n: 3 Summe: 4
     #<continuation> n: 2 Summe: 3
     2
```

2.5.3 Syntaxverbesserung durch Makros

Ein Makro ist ein Konstruktionsmittel zur Verbesserung der Syntax. Der Begriff *Makro* entstand im Rahmen der maschinennahen (Assembler-)-Programmierung. Kommen Zeilen, z. B. Anweisungen, Befehle oder Deklarationen, in einer bestimmten Reihenfolge mehrmals vor, so ist es zweckmäßig, diese „Einheit“ eindeutig und mnemotechnisch geschickt zu benennen. Im Quellcodetext wird dann an den einzelnen Stellen nur der eindeutige Bezeichner notiert. Makrobefehle verkürzen den Text wesentlich. Erst bei der Interpretation bzw. Compilation werden Makros aufgelöst, d. h. der jeweils ersetzte Text regeneriert. Diese sogenannte Makroexpansion ist eine rein textuelle Umformung.

In LISP sind zwei Arten von Makros zu unterscheiden: Makros und *READ*-Makros. „Normale“ Makros expandieren in der `EVAL`-Phase. *READ*--Makros expandieren in der *READ*-Phase des *READ-EVAL-PRINT*-Zyklus.

Das Hochkomma als Kurzschreibweise für das `quote`-Konstrukt (↪ Abschnitt 1.1.3 S. 27) ist eines von zahlreichen eingebauten *READ*-Makros. In mächtigen LISP-Systemen (z. B. in *Common LISP*) kann der Benutzer selbst *READ*-Makros definieren. Wir konzentrieren uns im folgenden auf die allgemein verfügbaren *READ*-Makros `quasiquote` (Kurzzeichen: ` — Achtung! Backquotezeichen ≡ „rückwärtiges“ Hochkomma), `unquote` (Kurzzeichen: `,`) und `unquote-splicing` (Kurzzeichen: `@`). Als Beispiel für die Verbesserung der Übersichtlichkeit durch Makros dient das Erweitern oder Modifizieren einer bestehenden Umgebung im Rahmen einer Kontenverwaltung.

Wir definieren einen Namensraum (Umgebung) `Konten-Management` mit dem `make-base-namespace`-Konstrukt (↪ S. 42; Näheres ↪ Abschnitt 2.7 S. 319).

```
eval> (define Konten-Management
        ((lambda ()
           (make-base-namespace))))
eval> Konten-Management ==> #<namespace:0>
```

Den Namensraum `Konten-Management` ergänzen wir um die Symbole `Foo` und `Baz`.

```
eval> (eval '(define Foo '(+ 1 2))
            Konten-Management)
eval> (eval 'Foo Konten-Management)
      ==> (+ 1 2)
eval> (eval '(define Baz Foo)
            Konten-Management)
eval> (eval 'Baz Konten-Management)
      ==> (+ 1 2)
```

Ist ein Teil des ersten Argumentes von `eval` jedoch zu evaluieren, hier `(+ 1 2)`, dann ist mit Hilfe des `list`-Konstruktes erst ein auswertungsfähiges Argument zu konstruieren.

```
eval> (eval '(list 'define 'Baz Foo)
            Konten-Management)
      ==> (define Baz (+ 1 2))
```

oder

```
eval> (eval (eval '(list 'define 'Baz Foo)
                  Konten-Management)
            Konten-Management)
eval> (eval 'Foo Konten-Management)
      ==> (+ 1 2)
eval> (eval 'Baz Konten-Management)
      ==> 3
```

Bei einem umfangreicheren Argument wirken die dann erforderlichen vielen `list` und `quote` Formulierungen wenig übersichtlich. Eine syntaktische Vereinfachung bietet `quasiquote` in Verbindung mit dem `unquote`-Konstrukt. Die Kurzschreibweise für das `quasiquote`-Konstrukt ist das Backquotezeichen. Die Kurzschreibweise für das `unquote`-Konstrukt ist das Komma.

```
eval>  (eval `(define Baz ,
                (eval 'Foo Konten-Management))
        Konten-Management)
eval> (eval 'Baz Konten-Management) ==> 3
```

Innerhalb des `quasiquote`-Konstruktes hebt ein `unquote`-Konstrukt die Wirkung als `quote`-Konstrukt auf, d. h. der symbolische Ausdruck nach dem Komma ist nicht vom `quote` betroffen. In manchen Fällen ist das Einspleißen (engl.: *splicing*) eines Wertes erforderlich. Eine Liste ist als Folge ihrer Elemente, also ohne Klammern in eine andere Liste einzufügen. Dazu dient das `unquote-splicing`-Konstrukt. Es ist nur in Zusammenhang mit einem `unquote`-Konstrukt anwendbar. Man beachte, dass unmittelbar nach dem Komma das Zeichen `@` (spöttisch „Klammeraffe“) notiert ist.

```
eval> ;;ASCII
      ;; (American Standard Code for
      ;; Information Intergage)
      (integer->char 64) ==> #\@
eval> (display (integer->char 64))
      ==> @
```

Wir zeigen die Wirkungsweise anhand von ein paar Beispielen:

```
eval> (define Bar '(0 1 2 3 4))
eval> (define Baz '(5 6))
eval> (cons '+ (append Bar Baz))
      ==> (+ 0 1 2 3 4 5 6)
eval> '(+ ,@Bar ,@Baz) ;Kein Backquotezeichen
      ==> (+ ,@Bar ,@Baz)
```

```
eval> `(+ ,@Bar ,@Baz)
      ==> (+ 0 1 2 3 4 5 6)
eval> `(+ ,@ Bar ,@      Baz)
      ==> (+ 0 1 2 3 4 5 6)
eval> `(+ ,@ Bar , @ Baz)
       ==> ERROR ...
      ;Symbol @ nicht definiert in  aktueller
      ; Umgebung. Achtung! Leerzeichen zwischen
      ; Komma und "Klammeraffen" macht diesen
      ; zu einem selbständigen Symbol.
```

Mit dem `quasiquote`-Konstrukt drehen wir die übliche LISP-Notation um. Jetzt wird kein Ausdruck ausgewertet, der nicht besonders gekennzeichnet ist (durch das `unquote`-Konstrukt). Normalerweise wird jeder Ausdruck ausgewertet, es sei denn, er ist besonders (durch das `quote`-Konstrukt) gekennzeichnet.

Ein Makro, der in der *EVAL*-Phase expandiert, kann am einfachsten mit dem `define-syntax-rule`-Konstrukt erzeugt werden. Dabei sei angemerkt, dass solche Makros eigentlich „*un-Schemely*“ sind, weil sie Nebeneffekte bezüglich von Variablen ermöglichen. Der Zweck von solchen Makros ist die Möglichkeit eine von anderen Sprachen her gewohne Syntax zu schaffen — deshalb auch der Name dieses Konstruktes.

Das `define-syntax-rule`-Konstrukt bindet einen Makro, der ein Muster (engl.: *pattern*) in die Vorlage (engl.: *template*) einarbeitet.

`(define-syntax-rule (`< *pattern* >`) (`< *template* >`))`

mit:

< *pattern* >	≡ Das Muster beginnt stets mit einem Identifier. Nach diesem Initialidentifier sind die folgenden Identifier Variablen (*Macro Pattern Variablen*), die bei der Makroanwendung im Hinblick auf das „Passen“ (*match*) im < *template* > genutzt werden.
< *template* >	≡ Ein Konstrukt, das zum Muster passt und zwar in der Art, das jede passende Variable in der Vorlage durch den Wert der Variablen ersetzt wird.

Der folgende Makro `Tausch` verdeutlicht die Wirkungsweise des `define-syntax-rule`-Konstruktes (↪ Abbildung A.2 S. 469).

```
eval> (define-syntax-rule (Tausch x y)
        (let ((Temp x))
          (set! x y)
          (set! y Temp)
          (list x y)))
eval> (define First 'First)
eval> (define Second 'Second)
eval> (define Foo (Tausch First Second))
eval> Foo ==> (Second First)
```

```
eval> First ==> Second
eval> Second ==> First
eval> (define Foo (Tausch First Second))
eval> Foo ==> (First Second)
eval> First ==> First
eval> Second ==> Second
```

Das `define-syntax-rule`-Konstrukt deklariert einen Namen als spezielles Schlüsselwort. Ist beim Evaluieren ein solches Schlüsselwort in funktionaler Stellung, d. h. erstes Element einer Liste, dann wird zunächst die Expandierungsvorschrift ausgewertet und anschließend das entstandene Resultat evaluiert.

```
eval> (define-syntax-rule (Mycons x)
        (lambda (Sexpr)
          `(cons ,(x Sexpr 1)
                 ,(x Sexpr 0))))
eval> (define Foo (Mycons list-ref))
```

Beispiel: Makroexpansion `Mycons`

Der Makro `Mycons` expandiert für dieses `Foo`-Konstrukt wie folgt:

```
(define Foo
  (lambda (Sexpr)
   (quasiquote
     (cons (unquote (list-ref Sexpr 1))
           (unquote (list-ref Sexpr 0))))))
```

Mit dieser Makroexpansion, auch als Makrotransformation bezeichnet, lässt sich das Ergebnis des Evaluierens von `Foo` leichter nachvollziehen.

```
eval> (Foo '(a b c))
      ==> (cons b a)
```

Der Makro `Mycons` wird nochmals wie folgt genutzt:

```
eval> (define Bar (Mycons +))
```

Seine Makrotransformation ergibt mit der Kurzschreibweise:

```
(define Bar
 (lambda (Sexpr) `(cons ,(+ Sexpr 1) ,(+ Sexpr 0)))))
```

Damit ergibt sich z. B. :

```
eval> (Bar 6) ==> (cons 7 6)
eval> (define Baz
       (lambda (Anfang Rest)
        ((Mycons list-ref) (list Anfang Rest))))
eval> (Baz 'Heute 'Arbeiten)
      ==> (cons Arbeiten Heute)
```

Die Makro-Option ist einerseits ein mächtiges Konstruktionsmittel, andererseits in der Praxis eine häufige Fehlerquelle. Sorgfältig zu beachten sind der Zeitpunkt und die Umgebung des Textersatzes. Makros sind stets vor ihrer ersten Nennung in einer Funktion zu definieren, wie das folgende Beispiel zeigt:

```
eval> (define Foo
        (lambda (x)
          (2* x)))
eval> (define-syntax-rule (2* Multiplikant)
        (eval `(* 2 ,Multiplikant)))
eval> (Foo 4) ==> ERROR ...
      ;reference to an identifier before
      ; its definition: 2*

eval> (2* 5) ==> 10
eval> (define Foo
        (lambda (x)
          (2* x)))
eval> (Foo 4) ==> 8
```

Um Verwirrungen zu vermeiden, sollten wir einem Symbol als Schlüsselwort einer Makrodefinition nicht einen anderen Wert zuweisen.

```
eval> (define 2* +)
eval> (Foo 7) ==> 14
eval> (define Baz
        (lambda (x)
          (2* x)))
eval> (Baz 7) ==> 7
```

Eine etwas umfangreichere Makrodefinition zeigt das Programm 2.8 S. 298. Das dortige `definep`-Konstrukt, eine Funktion, präziser ein `#<procedure>`-Konstrukt, ist in der Lage, ähnlich wie ein `let`-Konstrukt (↪ Abschnitt 1.2.2 S. 69), beliebig viele Symbole an Werte zu binden. Ein wesentliches Kriterium ist die jeweilige Umgebung, in der das einzelne `define`-Konstrukt evaluiert wird. Erst wird dafür der Wert ermitteln und zwar in der aktuellen Umgebung (vor Eintritt in das Konstrukt), hier `global-namespace` genannt. Anschließend wird das jeweilige `define`-Konstrukt in der inneren Umgebenung des `let`-Konstruktes, hier `let-namespace` genannt, evaluiert. Die beiden Umgebungen sind zu unterscheiden und sind daher beim `eval`-Konstrukt explizit anzugeben (Näheres dazu ↪ Abschnitt 2.7 S. 319). Die Lösung hat folgende Grundstruktur:

```
(define global-namespace (current-namespace))
(let ((let-namespace (make-base-namespace)))
 (begin
   (eval `(define ,< variable_i >
            ,(eval < value_i > global-namespace))
```

```
eval> (define-syntax-rule (definep L) (lambda ()
      (let* ((let-namespace
                (make-base-namespace))
             ;;Falls keine vollständigen
             ;; Symbol-Wert-Paare
             (Liste (cond
                      ((even? (length L))
                        L)
                      (#t (append L
                            (list #f)))))
             (j (length Liste)))
        (begin
          (do ((i 0 (+ i 2))) ((>= i j))
            (eval `(define ,(list-ref Liste (+ i 0))
                     ,(eval (list-ref Liste (+ i 1))
                        global-namespace))
              let-namespace))
          let-namespace))))
```

Legende:
Das `definep`-Konstrukt dient zum Binden von mehreren Symbolen.

Tabelle 2.8: Programm: `definep`

```
                            let-namespace))
 let-namespace) ;Rückgabewert
```

Anhand einiger Beispiele verdeutlicht das `definep`-Konstrukt die Werte in verschiedenen Namensräumen.

```
eval> (define global-namespace
              (current-namespace))
eval> (define Foo 10)
eval> (define ns
       ((definep '(Foo (* 2 Foo) Baz (+ 3 4) Bar))))
```

Beispiel: Makroexpansion `definep`

Der Makro `definep` (↪ S. 298) expandiert für dieses `ns`-Konstrukt wie folgt:

```
(define ns
   ((lambda ()
      (let* ((let-namespace
               (make-base-namespace))
```

```
            (Liste
             (cond
              ((even?
                (length
                  '(Foo (* 2 Foo) Baz (+ 3 4) Bar)))
               '(Foo (* 2 Foo) Baz (+ 3 4) Bar))
              (#t
               (append
                '(Foo (* 2 Foo) Baz (+ 3 4) Bar)
                (list #f)))))
            (j (length Liste)))
      (begin
        (do ((i 0 (+ i 2)))
            ((>= i j))
          (eval
           `(define ,(list-ref Liste (+ i 0))
              ,(eval
                (list-ref Liste (+ i 1))
                global-namespace))
           let-namespace))
        let-namespace))))))
```

Mit dieser Zwischenstufe bei der Makro-Ausführung werden die Werte von `ns`, `Foo` und leichter nachvollziehbar:

```
eval> ns ==> #<namespace:0>
eval> Foo ==> 10
eval> Baz ==> ERROR ...
      ;reference to an identifier
      ; before its definition: Baz
eval> (eval 'Foo ns) ==> 20
eval> Foo ==> 10
eval> (eval 'Baz ns) ==> 7
eval> (eval 'Bar ns)
       ==> #f ;Ersatzwert, da keiner
              ; angegeben wurde.
```

Bindet man im Programm 2.8 S. 298 das `let-namespace` statt an die Applikation von `make-base-namespace` an die Applikation von `current-namespace` erhält man folgende Ergebnisse[15]:

```
eval> (define global-namespace
               (current-namespace))
eval> (define Foo 10)
eval> (define ns
       ((definep '(Foo (* 2 Foo) Baz (+ 3 4) Bar))))
eval> Foo ==> 20
```

[15] In diesem Fall hätte man natürlich auf die explizite Nennung der Namensräume verzichten können, da das `eval`-Konstrukt standardmäßig im `current-namespace` ausgeführt wird.

```
eval> Baz ==> 7
eval> Bar ==> #f
```

2.5.4 Zusammenfassung: Continuation und Makros

Die Verknüpfung der Funktion mit ihrer Definitionsumgebung (*Closure*-Konzept) ermöglicht problemloses Konstruieren mit Funktionen höherer Ordnung.

Eine Funktion kann an ihre eigene `lambda`-Variable gebunden werden. Auf diese Weise entstehen selbstbezügliche Funktionen (ohne `define`- oder `letrec`-Konstrukt).

Das `call/cc`-Konstrukt bindet an die `lambda`-Variable seiner applizierten Funktion, die noch nicht vollzogene Abarbeitung als sogenannte *Continuation* („Fortsetzung"). Da eine *Continuation* die vollzogenen Bindungen mit umfasst, können Berechnungen abgebrochen und später (auch mehrfach) wieder aufgenommen werden. Das *call/cc*-Konstrukt ist eine Konstrollstruktur, die Sprünge in die „Zukunft" und „Vergangenheit" einer Berechnung ermöglicht.

Die Umgebung einer Funktion ist als ein lokaler Namensraum betrachtbar. Die hierarchische Anordnung solcher Namensräume ist eine Technik zur Vermeidung von Namenskonflikten.

Makros dienen zur Syntaxverbesserung. Zu unterscheiden sind „normale" Makros von *READ*-Makros. *READ*-Makros sind z. B. das `quasiquote` (Zeichen: `` ` ``), das `quote` (Zeichen: `'`), das `unquote` (Zeichen: `,`) und das `unquote-splicing` (Zeichen: `@`). Sie expandieren in der *READ*-Phase des *READ-EVAL-PRINT*-Zyklus. „Normale" Makros expandieren in der *EVAL*-Phase. Sie können, obwohl sie eigentlich „*un-Schemely*" wegen ihrer möglichen Nebeneffekte sind, mit dem Konstrukt `define-syntax-rule` definiert werden. Einerseits ist ein Makro ein mächtiges Konstruktionsmittel, andererseits verursacht ein Makro im Programmieralltag häufig Fehler, weil der Expanionszeitpunkt und die Expansionsumgebung nicht beachtet werden.

Charakteristische Beispiele für Abschnitt 2.5

Beispiel: Lexikalische Bindung

```
;;;;Funktion und Umgebung (lexikalische Bindung)
eval> (define Zaehler 5)
eval> (define Foo
        (lambda ()
          (set! Zaehler (+ 1 Zaehler))))
eval> (let ((Zaehler 0))
        (Foo)
        Zaehler) ==> 0
eval> Zaehler ==> 6
```

```
eval> (define Foo
        (lambda (Fkt x y)
          (let ((+ (lambda (x y) (apply +
                                   (list x y)))))
               (list (+ (Fkt x) (Fkt y))
                     (+ (Fkt (Fkt x))
                        (Fkt (Fkt y)))))))
eval> (define Baz
        (lambda (z) (+ z z)))
eval> (Foo Baz 1 2) ==> (6 12)

eval> (let ((Fkt (lambda (Fun x)
                   (cond ((zero? x) (Fun))
                         (#t (lambda () x))))))
        (Fkt (Fkt #f 6) 0)) ==> 6
```

Beispiel: Klassisches LISP-Sprung-Paar
Die beiden Konstrukte `catch` und `throw` sind im klassischen LISP das übliche Sprung-Paar. `catch` setzt die Marke; `throw` springt zur Marke. Wir realisiert hier dieses Sprung-Paar mit der allgemeinen Kontrollstruktur `call-with-current-continuation`, kurz: `call/cc`.

```
eval> (define  *CATCHER-CONTINUATIONS*  null)
eval> (define-syntax-rule (catch Liste)
        (lambda ()
          (let ((-C- (gensym)))
            `(call/cc
               (lambda (,-C- )
                    (set! *CATCHER-CONTINUATIONS*
                      (cons (list ,(cadr Liste) ,-C-)
                            *CATCHER-CONTINUATIONS*))
                    ,(cons `begin (cddr Liste)))
                      ))))
eval> (define throw
        (lambda (Marke Sexpr)
          (letrec ((Member-Restliste
                    (lambda (Key L)
                      (cond ((null? L) (list))
                            ((eq? Key (caar L)) L)
                            (#t
                             (Member-Restliste Key
                               (cdr L))))))
                   (Continuations
                     (Member-Restliste
                       Marke
                       *CATCHER-CONTINUATIONS*))
                   (Get-C- (lambda ()
                         (cadr (car Continuations)))))
```

```
            (cond ((null? Continuations)
                   (error "Keine catch-Marke: " Marke))
                  (#t (set! *CATCHER-CONTINUATIONS*
                            (cdr Continuations))
                      ((Get-C-) Sexpr))))))
```

Wir definieren nun ein simples Konstrukt `f1` und erhalten mit der Makroexpansion das folgende Äquivalent:

```
eval> (define f1  (catch '(+ 'Foo 2 3)))
eval> (define f1 (lambda ()
      (let ((-C- (gensym)))
        `(call/cc
          (lambda (,-C-)
            (set! *CATCHER-CONTINUATIONS*
              (cons
               (list ,(cadr '(+ 'Foo 2 3)) ,-C-)
               *CATCHER-CONTINUATIONS*))
            ,(cons `begin (cddr '(+ 'Foo 2 3)))))))))
eval> (eval f1) ==> #<procedure:f1>
eval> (f1) ==>
      (call/cc (lambda (g725)
                 (set! *CATCHER-CONTINUATIONS*
                   (cons (list 'Foo g725)
                         *CATCHER-CONTINUATIONS*))
                  (begin 2 3)))
eval> (begin (+ 1 5
                (eval ((catch '(+ 'Foo 2 3)))))
                      (throw 'Foo (+ 4 5)))
      ==> 15
```

Beispiel: Sich selbst reproduzierendes Konstrukt

```
eval> (define Selbst
        ((lambda (Sexpr)
           (list Sexpr Sexpr))
         (lambda (Sexpr)
           (list Sexpr Sexpr))))
eval> Selbst ==>
     (#<procedure:Sexpr>
      #<procedure:Sexpr>)
eval> (eval (eval (eval Selbst)))
      ==> (#<procedure:Sexpr>
           #<procedure:Sexpr>)
eval> (equal? Selbst (eval Selbst)) ==> #t
```

Beispiel: Berechnung der Länge einer Liste

```
eval> (define Compose
```

```
        (lambda (F G)
          (lambda (X)
            (F (G X)))))
eval> (define Conditional
        (lambda (P F G)
          (lambda (X)
            (if (P X) (F X) (G X)))))
eval> (define F-ADD
        (lambda (X) (apply + X)))
eval> (define Alpha
        (lambda (F)
          (lambda (X)
            (map F X))))
eval> (define F-Length
        (lambda (X)
          ((Conditional
            null?
            (lambda (X)
              0)
            (Compose F-ADD
                     (Alpha
                      (lambda (X) 1))))
           X)))
eval> (F-Length '(a b c)) ==> 3

;;;Ersetzt man die Konstrukte
;;; Conditional, Compose, Alpha
;;; und F-ADD, dann ergibt sich
;;; folgende Definition für
;;; das F-Length-Konstrukt:
eval> (define F-Length
        (lambda (X)
          (((lambda (P F G)
              (lambda (X)
                (if (P X) (F X) (G X))))
            null?
            (lambda (X)
              0)
    ((lambda (F G)
       (lambda (X)
         (F (G X))))
     (lambda (X)
       (apply + X))
           (lambda (X)
             (map (lambda (X) 1) X))))
           X)))
eval> (F-Length '(a b c d)) ==> 4
```

Beispiel: Abwechselnde Auführung zweier Aufgaben Mit einer abwechselnden Abarbeitung der Teilaufgaben von zwei „Prozessen“ wird die gemeinsame Nutzung von Makro und Funktion verdeutlicht. Die noch abzuarbeitenden Teilaufgaben sind als `lambda`-Konstrukt an das Symbol `*PR*` gebunden.

```
eval> (define *PR* null)
eval> (define Process-Stack
        (lambda (Task_List)
          (lambda ()
            (cond
             ((null? Task_List) null)
             (#t
              (begin0
                (car Task_List)
                (set! Task_List (cdr Task_List))))))))
eval> (define Alternate
        (lambda (Execute_Process Wait_Process)
          (lambda ()
            (let ((Activ Execute_Process))
              (set! Execute_Process Wait_Process)
              (set! Wait_Process Activ)
              (Activ)))))
eval> (define Execution-Order
        (lambda (n)
          (letrec ((step
                    (lambda (m)
                      (cond
                       ((> m n) 'stop)
                       (#t
                        (display m)
                        (display ". step: ")
                        (display (*PR*))
                        (newline)
                        (step (+ m 1)))))))
            (step 1))))
eval> (define-syntax-rule
       (Process-Rotation Liste-1 Liste-2)
         (Alternate
          (Process-Stack 'Liste-1)
          (Process-Stack 'Liste-2)))
eval> (define-syntax-rule
       (Task-Lists L1 L2)
        (set! *PR* (Process-Rotation L1 L2)))

eval> (Task-Lists (Task-A Task-B Task-C)
                  (Task-I Task-II Task-III Task-IV))
```

Entspricht nach Makroexpansion der Ausführung von:

```
eval> (set! *PR*
```

```
       (Alternate
        (Process-Stack
         '(Task-A Task-B Task-C))
        (Process-Stack
         '(Task-I Task-II Task-III Task-IV))))

eval> (Execution-Order 4) ==>
1. step: Task-A
2. step: Task-I
3. step: Task-B
4. step: Task-II
stop
eval> (Execution-Order 2) ==>
1. step: Task-C
2. step: Task-III
stop
eval> (Execution-Order 3) ==>
1. step: ()
2. step: Task-IV
3. step: ()
stop
```

Beispiel: Makro/Funktions-Gegenüberstellung

```
eval> (define Foo 6)
eval> (define Call-by-Value-Demo
        (lambda (X)
          (set! X (* X X))
          (- X Foo)))
eval> (Call-by-Value-Demo Foo)
      ==> 30
eval> Foo ==> 6
eval> (define-syntax-rule
       (Call-by-Reference-Demo Sexpr)
         (begin
             (set! Sexpr (* Sexpr Sexpr))
                  (- Sexpr Foo)))
eval> (define Baz 6)
eval> (Call-by-Reference-Demo Baz)
      ==> 30
eval> Baz ==> 36
eval> Foo ==> 6
eval> (Call-by-Reference-Demo Foo)
      ==> 0 ;Achtung!
            ; Name der freien Variablen
            ; benutzt.
eval> Foo ==> 36
```

2.6 Generierte Zugriffskonstrukte

Dieser Abschnitt behandelt das automatische Generieren von Zugriffskonstrukten. Für die Liste, die Assoziationsliste (A-Liste) oder die Eigenschaftsliste, (P-Liste) konnten wir die primitiven, eingebauten Konstruktoren, Selektoren, Prädikate und Mutatoren nutzen, um damit unseren anwendungsspezifischen Verbund zu konstruieren. War auf ein bestimmtes Feld einer A-Liste zuzugreifen, z. B. auf das Feld `Name-des-Kontoinhabers`, dann konnte mit dem `assq`-Konstrukt ein entsprechender Selektor, wie folgt, definiert werden (↪ Abschnitt 2.2.2 S. 183).

```
eval> (define Name-des-Kontoinhabers
        (lambda(A_Liste)
          (assq 'Name-des-Kontoinhabers
                A_Liste)))
```

Neben den Selektoren sind die Prädikate und Mutatoren ebenfalls aus den eingebauten Grundfunktionen selbst zu konstruieren. Solche Definitionen können direkt vom LISP-System generiert werden. Zum Zeitpunkt der Definition einer Datenstruktur sind unter Angabe einiger Steuerungsparameter die Konstrukte des Verbundes generierbar. Dazu dient das `define-struct`-Konstrukt (↪ Abschnitt 2.6.2 S. 309).

Vorab betrachten wir das Konzept der „passiven“ Daten primär aus der Sicht „aktiver“ Zugriffskonstrukte. Erörtert wird das Konzept abstrakter Datentypen (↪ Abschnitt 2.6.1 S. 306). Wesentlicher Vorteil ist die Rückgriffsmöglichkeit auf existierende Datentypen bei der Definition neuer Datentypen. Es werden Eigenschaften von existierenden Datentypen auf neue „vererbt“. Wir zeigen die einfache statische Vererbung (↪ Abschnitt 2.6.3 S. 314). Eine weitergehende Vererbung wird im Zusammenhang mit Klassen-Instanz-Konzepten behandelt (↪ Abschnitt 2.8.2 S. 337).

2.6.1 Abschirmung von Werten

Definieren wir ein Konstrukt, dann bemühen wir uns mit den gewählten Symbolen und Strukturen eine gedankliche Verknüpfung zwischen realen und formalen „Einheiten“ (Objekten) herzustellen. Entsprechend dem jeweiligen Denkrahmen (Paradigma, ↪ Abschnitt 2.1 S. 141) und den Möglichkeiten seiner Umsetzung betonen wir eine „passive“ (daten-/-wert-bezogene) oder „aktive“ (operations-bezogene) Sicht (↪ Abschnitt 1.2 S. 50).

Lösen wir eine kleine Programmieraufgabe wie das Verwalten einigen Postanschriften, dann werden wir im imperativ-geprägten Denkrahmen frühzeitig die benötigten Speicherplätze benennen und ordnen. Wir definieren eine passive Datenstruktur. Eine solche Datenstruktur zeigt Tabelle 2.9 S. 307. Sie ist in COBOL, einer der Programmiersprache für das imperativ-geprägte Paradigma, formuliert.

```
01   ADRESSE.
     05   NAME.
          10   ZUNAME          PICTURE X(30).
          10   VORNAME         PICTURE X(30).
          10   TITEL           PICTURE X(10).
     05   WOHNORT.
          10   STRASSE         PICTURE X(20).
          10   POSTLEITZAHL    PICTURE 9(4).
          10   ORT             PICTURE X(30).
     05   KOMMUNIKATION.
          10   TELEFON         PICTURE X(15).
          10   TELEFAX         PICTURE X(15).
          10   E-MAIL          PICTURE X(15).
     05   SONSTIGES.
          10   STOCKWERK       PICTURE 99.
          10   POSTFACH        PICTURE 9(5).
```

Legende:

01...10	≡ Gruppennummern zur Strukturierung.
PICTURE X(30)	≡ Feld für 30 alphanumerische Zeichen.
PICTURE 9(4)	≡ Feld für eine Zahl mit 4 Ziffern.

Tabelle 2.9: Beispiel: *COBOL*-Datendefinition

Dabei haben wir früh Angaben über die Wertbereiche der einzelnen Felder bereitgestellt. Aufgrund der PICTURE-Angabe ist die Variable ZUNAME maximal 30 Zeichen lang, während die POSTLEITZAHL eine Zahl mit 4 Ziffern ist. Der Wertebereich eines Feldes ist hier mit der PICTURE-Klausel durch eine Längenangabe und einen Verweis auf einen eingebauten Datentyp definiert.

Mit den Namen dieser Datenstruktur werden Schritt für Schritt benötigte Operationen durchdacht und spezifiziert. Eine Operation Drucke--Adress-Aufkleber enthält dann beispielsweise eine Formulierung: „Bilde-Druckzeile aus TITEL, VORNAME und ZUNAME. Falls die Zeilenlänge gößer als 25 Zeichen ist, dann nimm vom Feld TITEL die ersten drei Zeichen ... .“ Jetzt wird erkannt, dass man eine Operation KURZ-TITEL, die vom Feld TITEL die ersten drei Zeichen selektiert, benötigt. Im Kontext „passive“ Datenstruktur bietet sich eine Änderung der TITEL-Definition, wie folgt, an:

```
10 TITEL.
   15 KURZTITEL          PICTURE XXX.
   15 RESTTITEL          PICTURE X(7).
```

Mit dieser modifizierten TITEL-Definition kann man TITEL oder KURZTITEL drucken. Als Alternative wäre die Definition einer Operation Kurztitel möglich, die im *COBOL*-Programm auf einer MOVE-Operation (*assign statement*) beruhen würde.

Im funktions-geprägten Denkrahmen benennen wir frühzeitig Operation. Ihr schrittweises durchdenken und spezifizieren führt schließ-

lich ebenfalls zu den abzubildenden „passiven“ Einheiten (Daten). Wir hätten beispielsweise eine Funktion `Adress-Aufkleber` mit einem Argument `ADRESSE` definiert. Beim Konstruieren des Nebeneffektes `print` benötigen wir die Operationen `Get-Titel` und `Get-Kurztitel`. Auch jetzt können wir wieder für `Get-Kurztitel` eine operationale Sicht umsetzten, d. h. in LISP ein `lambda`-Konstrukt wählen. Dieses enthält die Operationen `Get-Titel` und eine Verkürzungsoperation. Diese ist abhängig vom Typ des Wertes, den `Get-Titel` liefert. Ist es ein String, dann käme zum Verkürzen das `substring`-Konstrukt in Betracht (↪ Abschnitt 2.4.2 S. 241). Ist es ein Symbol, wäre ein `Explode`-Konstrukt notwendig (↪ Abschnitt 2.4.3 S. 248).

Im imperativ-geprägten Denkrahmen wurden frühzeitig die Wertebereiche der Felder beschrieben. Wir haben in den Kategorien wie Länge und primitiven Beschreibungstyp der Wertebereichsspezifikation gedacht. Holzschnittartig formuliert: Die Speicherplatzkonditionen prägen diese Denkwelt. Ausgangspunkt für die Problemdurchdringung ist die Frage: Wie werden die anwendungsspezifischen Daten repräsentiert?

Im funktions-geprägten Denkrahmen kannten wir zum Zeitpunkt der Operation `Adresse-Aufkleber` weder die Länge des Feldes noch musste schon der Wertebereich mit eingebauten bzw. bekannten Datentypen präzisiert werden. Erst mit der Definition von `Get-Kurztitel` sind Angaben über den Wertebereich entsprechend zu präzisieren. Mit einem Denken in der Kategorie Zugriffskonstrukt verschieben wir eine solche, implementationsabhängige Beschreibung. Wir können zunächst davon abstrahieren, ob die Daten z. B. vom Typ String, Vector oder Symbol sind. Wir durchdenken und spezifizieren unsere Konstruktion in der Kategorie Zugriffskonstrukte. Ausgangspunkt für die Problemdurchdringung ist die Frage: Welche anwendungsspezifischen Operationen sind mit Daten notwendig?

Beide Ansätze sind voneinander abhängig. Immer dann, wenn ein Problem wenig durchschaut wird, ist ein Wechselspiel zwischen beiden Ansätzen zur Korrektur der ersten Lösungsentwürfe erforderlich. Änderungen der „passiven“ Datenstruktur werden durch Änderungen der „aktiven“ Konstrukte (Operationen) hervorgerufen und umgekehrt. Die Definition von Zugriffskonstrukten bildet dabei eine Schnittstelle zwischen der Implementations- und der Verwendungs-Perspektive. Mit den Zugriffskonstrukten versuchen wir die Implementations-Anforderungen (Wertebeschreibungen in den Kategorien der eingebauten Konstrukte) von den Verwendungsanforderungen (anwendungsspezifischen Operationen) abzuschirmen.

In LISP kann diese Schnittstelle „verheimlichen“, ob nur auf die Wertzelle eines Symbols zugegriffen oder ein `lambda`-Konstrukt appliziert wird. Im ersten Fall haben wir es mit dem Zugriff auf ein „passives“ Datum zu tun. Im zweiten Fall geht es um eine Funktionsanwendung, d. h. um ein „aktives“ Datum.

Beide Arten von Zugriffskonstrukten sind mit der Datenstrukturdefinition automatisch generierbar. Das `define-struct`-Konstrukt erzeugt „passive“ Zugriffskonstrukte (↪ Abschnitt 2.6.2 S. 309). Das Generieren von „aktiven“ Zugriffskonstrukten („*aktiven Slots*“) ist Bestandteil des Klassen-Instanz-Konzeptes.

Zum Verständnis des `define-struct`-Konstruktes betrachten wir erneut unsere Kontenverwaltung (↪ Abschnitt 2.7 S. 319). Dabei konstruieren wir die neue Struktur `Verwahrkonto` aus der definierten Struktur `Konto`. Verdeutlicht wird damit der Vererbungsmechanismus des `define-struct`-Konstruktes (↪ Abschnitt 2.6.3 S. 314).

2.6.2 `define-struct`-Konstrukt

Mit dem `define-struct`-Konstrukt (auch `defstruct` genannt, z. B. in *Common LISP* oder `define-structure` in *PC Scheme*) definieren wir eine (Datensatz-)Struktur aus Feldern (engl.: *slots*) mit gegebenenfalls voreingestellten Werten (engl.: *default values*) gemeinsam mit ihren Selektoren, Prädikaten und Mutatoren. Das `define-struct`-Konstrukt ermöglicht den Zugriff auf bereits definierte Struturen. Neue Strukturen lassen sich aus bestehenden zusammenbauen. Vorteilhaft ist dabei, dass jeweils die Operationen für das Zugreifen und Ändern der Feldwerte unmittelbar vom LISP-System bereitgestellt werden. Sie müssen nicht erst selbst definiert werden.

Das `define-struct`-Konstrukt hat vielfältige Optionen. Ohne Vererbungsoption und mit etwas Vereinfachung hat es die folgende Syntax. Die Vererbungsoption behandeln wir anschließend (↪ Abschnitt 2.6.3 S. 314).

`eval>` `(define-struct` $<structure-name>$
$(\{<slot>\{<slot-option>\}\})$
$\{<struct-option>\})$

mit:

$<structure-name>$ ≡ Ein Symbol, das die Struktur benennt.
$<slot>$ ≡ Ein Symbol, das das Feld benennt.

```
<slot-option>   ≡ #:mutable
                | #:auto
<struct-option> ≡ #:mutable
                | #:super super-expr
                | #:inspector inspector-expr
                | #:auto-value auto-expr
                | #:guard guard-expr
                | #:property prop-expr val-exr
                | #:transparent
                | #:prefab
                | #:omit-define-syntaxes
                | #:omit-define-values
```

Als Wunscheffekt werden bei der Applikation des `define-struct`-Konstruktes folgende Operatoren generiert:

- Ein Konstruktor: `make-`$<structure-name>$
- ein Prädikat: $<structure-name>$?,
- für jedes Feld einen Selektor: $<structure-name>-<slot-name>$ und
- für jedes Feld einen Mutator, d. h. die Möglichkeit seinen Wert mit Hilfe des set!-Konstruktes zu ändern.
  (`set-`$<structure-name>-<slot-name><structure-name><new-value>$)

Für unsere Kontenverwaltung (↪ Abschnitt 2.7 S. 319) formulieren wir folgende Struktur:

```
eval> (define-struct Konto
        (Identifizierung
        Name-Inhaber
        Adresse-Inhaber
        (Einzahlungen #:auto)
        (Auszahlungen #:auto)
        (Kontostand #:auto))
        #:transparent
        #:mutable
        #:auto-value 0)
```

Konstruktor: Mit Hilfe des Konstruktors `make-Konto` legen wir die beiden Konten `A001` und `A002` an. Die generierte `make-Konto`-Funktion initialisiert die mit dem Keyword `#:auto` markierten Slots mit dem hinter `#:aut-value` angegebenen Wert.

```
eval> (define A001
        (make-Konto
          1
          "Leuphana Universität"
          "D-21339 Lüneburg"))
eval> (define A002
        (make-Konto
          2
          "Bonin"
          "D-21391 Reppenstedt"))
```

Das `make-Konto`-Konstrukt erzeugt einen eigenen Datentyp *Structure*, der mit dem Prädikat `struct?` erkannt wird. Dieser `struct`-Datentyp lässt sich in einen Vektor konvertieren und dann entsprechend weiter behandeln.

```
eval> A001 ==>
      #(struct:Konto
         1 "Leuphana Universität" "D-21339 Lüneburg"
                                                0 0 0)
eval> (struct? A001) ==> #t
eval> (struct->vector A001) ==>
        #(struct:Konto
          1 "Leuphana Universität" "D-21339 Lüneburg"
                                                  0 0 0)
eval> (vector-ref (struct->vector A001) 2)
      ==> "Leuphana Universität"
```

Prädikat: Das generierte Prädikat `Konto?` erkennt Objekte, die mit dem `make-Konto`-Konstrukt erzeugt wurden.

```
eval> (Konto? A001) ==> #t
eval> (Konto? (struct->vector A001)) ==> #f
```

Selektor: Für den Zugriff auf die einzelnen Felder hat das `define-struct`-Konstrukt Selektoren generiert. Ihr Name setzt sich aus dem Präfix Strukturnamen, gefolgt von einem Bindestrich und dem Slotnamen (Feldnamen), zusammen.

```
eval> (Konto-Adresse-Inhaber A001)
      ==> "D-21339 Lüneburg"
eval> (Konto-Adresse-Inhaber A002)
     ==> "D-21391 Reppenstedt"
```

Diese generierten Selektoren benutzen wir zum Definieren der Funktion `Describe-Konto`. Sie stellt die aktuellen Werte eines Kontos dar.

```
eval> (define Describe-Konto
        (lambda (K)
          (cond((Konto? K)
```

```
                    (newline)
                    (display "______ Inhalt des Kontos: ")
                    (display (Konto-Identifizierung K))
                    (display   " ______")
                    (newline)
                    (display "Name-Inhaber:    ")
                    (display (Konto-Name-Inhaber K))
                    (newline)
                    (display "Adresse-Inhaber: ")
                    (display (Konto-Adresse-Inhaber K))
                    (newline)
                    (display "Einzahlungen:    ")
                    (display (Konto-Einzahlungen K))
                    (newline)
                    (display "Auszahlungen:    ")
                    (display (Konto-Auszahlungen K))
                    (newline)
                    (display "Kontostand:      ")
                    (display (Konto-Kontostand K)))
                  (#t (error "Kein Konto: " K)))))

eval> (Describe-Konto A001) ==>

______ Inhalt des Kontos: 1 ______
Name-Inhaber:    Leuphana Universität
Adresse-Inhaber: D-21339 Lüneburg
Einzahlungen:    0
Auszahlungen:    0
Kontostand:      0
```

Mutator: Für die Änderung der Werte der einzelnen *Slots* (Felder) hat das `define-struct`-Konstrukt Mutatoren generiert. Der Name setzt sich wie folgt zusammen:

`set-`$<$ *structur − name* $>$`-`$<$ *slot* $>$`!`

Z. B. : `set-Konto-Einzahlungen!`

Die folgenden Konstrukte zum Buchen von Ein- und Auszahlungen unterscheiden sich gegenüber der Lösung in Abschnitt 2.7 S. 319 nur durch die generierten Selektoren und die zugehörenden Mutatoren. Sie bedürfen daher keiner besonderen Erläuterung.

```
eval> (define Betrag?
        (lambda (X)
          (and (number? X) (>= X 0))))
eval> (define Buchungsdatum?
```

```
        (lambda (X)
          (and (number? X) (>= X 100101))))
eval> (define Add-Einzahlung!
        (lambda (K E_Datum Betrag)
          (cond
            ((and (Konto? K)
              (Buchungsdatum? E_Datum)
              (Betrag? Betrag))
              (set-Konto-Einzahlungen! K
                  (cons (list E_Datum Betrag)
                         (Konto-Einzahlungen K)))
              (set-Konto-Kontostand! K
                  (+ Betrag (Konto-Kontostand K)))
              (Konto-Kontostand K))
            (#t (error
                 "Keine Einzahlungen hinzugefuegt! "
                  K E_Datum Betrag)))))
eval> (Add-Einzahlung! A001 100117 50.00)
      ==> 50.0
eval> (Add-Einzahlung! A001 100118 60.00)
      ==> 110.0
eval> (Describe-Konto A001)
      ==>

______ Inhalt des Kontos: 1 ______
Name-Inhaber:    Leuphana Universität
Adresse-Inhaber: D-21339 Lüneburg
Einzahlungen:    ((100118 60.0) (100117 50.0) . 0)
Auszahlungen:    0
Kontostand:      110.0

eval> (define Add-Auszahlung!
        (lambda (K A_Datum Betrag)
          (cond
            ((and (Konto? K)
                  (Buchungsdatum? A_Datum)
                  (Betrag? Betrag))
             (cond ((<= Betrag (Konto-Kontostand K))
                    (set-Konto-Auszahlungen! K
                        (cons (list A_Datum Betrag)
                               (Konto-Auszahlungen K)))
                    (set-Konto-Kontostand! K
                        (- (Konto-Kontostand K) Betrag))
                    (Konto-Kontostand K))
                   (#t (error "Keine Deckung fuer: "
                         Betrag))))
            (#t (error
                 "Keine Einzahlungen hinzugefuegt! "
                  K A_Datum Betrag)))))
```

```
eval> (Add-Auszahlung! A001 100119 120.00)
      ==> . . Keine Deckung fuer:  120.0
eval> (Add-Auszahlung! A001 100119 80.00)
      ==> 30.0
eval> (Describe-Konto A001)
      ==>

______ Inhalt des Kontos: 1 ______
Name-Inhaber:    Leuphana Universität
Adresse-Inhaber: D-21339 Lüneburg
Einzahlungen:    ((100118 60.0) (100117 50.0) . 0)
Auszahlungen:    ((100119 80.0) . 0)
Kontostand:      30.0
```

Wir nehmen an, dass neben den Konten der bisherigen Kontenstruktur andere Konten mit den beiden zusätzlichen Eigenschaften `Buchungsgrund` und `Bearbeiter` benötigt werden. Ein solches erweitertes Konto sei z. B. ein Verwahrkonto, d. h. ein Konto auf dem dubiose Vorgänge zwischengebucht werden. Zur Definition der Struktur `Verwahrkonto` sind nicht alle Felder (*slots*) neu zu beschreiben. Die Struktur `Verwahrkonto` kann auf der definierten Struktur `Konto` aufbauen. Oder aus der entgegengesetzten Blickrichtung formuliert: Die Struktur `Verwahrkonto` erbt die Felder der Struktur `Konto`.

2.6.3 Einfache statische Vererbung

Vererbung ist die Übertragung von körperlichen und seelischen Merkmalen der Eltern auf die Nachkommen. Angelehnt an diese biologische Betrachtung wäre für LISP-Systeme die „Genetik“ von der „Klassifikation“ zu unterscheiden. Genetische Vererbung führt nicht zur Weitergabe jeder Eigenschaft eines Objektes. Holzschnittartig formuliert: „Wenn die Mutter eine schmale Nase hat, kann die Tochter eine breite haben.“ Die Klassifikation entspricht der Wiederanwendbarkeit einer Beschreibung beim Nachkommen (zur mathematischen Betrachtung der Vererbung ↪ z. B. [185]). Außerdem ist mit dem Begriff Vererbung noch ein Modus „letzter Wille/Testament“ assoziierbar. Dieser steht für (fast) völlig freies Weitergeben.

Der Mechanismus des `define-struct`-Konstruktes vererbt alle Felder mit ihren voreingestellten Werten. Er hat folgende Syntax, die gegenüber der oben spezifizierterten (↪ S. 309) um $< super-structure-name >$ erweitert ist:

```
eval> (define-struct
        (< structure-name >< super-structure-name >)
                                  ({< slot >{< slot-option >}})
```

$\{< struct - option >\})$

mit:

$< super - structure - name >$ ≡ Ein Symbol, das die Struktur benennt, von der geerbt wird.

Für das `Verwahrkonto` geben wir daher die Struktur `Konto` als $< super - structure - name >$ an.

```
eval> (define-struct (Verwahrkonto Konto)
        (Buchungsgrund
         Bearbeiter)
        #:transparent
        #:mutable)
```

Mit dem generierten Konstruktor `make-Verwahrkonto` definieren wir das Konto `V001`:

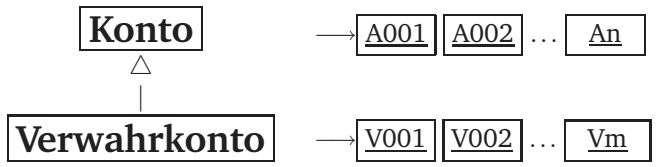
```
eval> (define V001
        (make-Verwahrkonto
           3
          "Meyer"
          "D-21399 Lüneburg"
          "Unbekannt"
          "Krause"))
eval> V001 ==>
       #(struct:Verwahrkonto
          3 "Meyer" "D-21399 Lüneburg" 0 0 0
          "Unbekannt" "Krause")
eval> (Verwahrkonto? V001) ==> #t
eval> (Konto? V001) ==> #t
```

Da ein Verwahrkonto, z. B. das Konto `V001`, alle Felder der Struktur `Konto` durch Vererbung aufweist, entspricht es sowohl der Struktur `Konto` wie der Struktur `Verwahrkonto`. Beide generierten Prädikate `Konto?` und `Verwahrkonto?` haben für `V001` den Wert `#t`. `V001` ist sowohl vom Typ `Konto` wie vom Typ `Verwahrkonto`. Wir haben mit dem `define-struct`-Konstrukt eine Typhierarchie konstruiert (↪ Abbildung 2.21 S. 316).

```
eval> (and (Konto? V001) (Verwahrkonto? V001)) ==> #t
```

Mit der Definition der Struktur `Konto` wurden die zugehörenden Selektoren (Zugriffsoperationen) generiert. Diese Zugriffsoperationen werden nicht „vererbt“. Wir können für Konten der Struktur `Verwahrkonto` die Selektoren der „Eltern“-Struktur `Konto` verwenden, allerdings nur unter der Bezeichnung der vererbenden Struktur, wie das folgende Beispiel zeigt:

```
eval> (Verwahrkonto-Bearbeiter V001)
      ==> "Krause"
```



Legende:

`Foo` $\lhd-$ `Bar`	$\equiv$	Erbungsrichtung, d. h. `Bar` erbt Felder und Operationen (Prädikate, Selektoren und Mutatoren) von `Foo`
$< struct >$	$\equiv$	Datenstruktur `Konto` bzw. `Verwahrkonto`
$< name >$	$\equiv$	Instanz der jeweiligen Datenstruktur `Konto` bzw. `Verwahrkonto`

Notation angelehnt an *Unified Modeling Language* (UML) (↪ Abschnitt 3.3 S. 418)

Abbildung 2.21: Skizze einer Vererbung von Datenstrukturen

```
eval> (Verwahrkonto-Name-Inhaber V001)
      ==> ERROR ...
      ;reference to an identifier
      ; before its definition:
      ; Verwahrkonto-Name-Inhaber
eval> (Konto-Name-Inhaber V001)
      ==> "Meyer"
```

In der Abbildung 2.21 S. 316 stellt der Pfeil mit der ausgeprägten Spitze einen gerichteten Grafen dar, in dem die Datenstruktur-Definitionen die Knoten (symbolisiert durch Rechtecke) sind. Die mit dem generierten Konstruktor (hier `make-Konto` oder `make-Verwahrkonto`) erzeugten Konten (z. B. `A001` oder `V001`) sind Ausprägungen der jeweiligen Datenstruktur. Sie sind als unterstrichene Namen in Rechtecken dargestellt. Solche Ausprägungen werden in der objekt-geprägten Terminologie als Instanzen eines (Daten-)Typs oder auch einer Klasse (hier der Datenstruktur) bezeichnet. Die jeweils definierten Slots (Felder) inklusive Feldwerten werden entlang der Pfeilrichtung geerbt.

Entsprechend dieser Vererbungshierarchie sind die Operationen der Struktur `Konto` auch auf Instanzen der Struktur `Verwahrkonto` anwendbar. Die Definition eines `Describe-Verwahrkonto`-Konstruktes kann daher auf dem obigen `Describe-Konto`-Konstrukt aufbauen. Es bedarf nur noch eines Zusatzes, der die beiden neuen Felder `Buchungsgrund` und `Bearbeiter` auf dem Bildschirm ausgibt.

```
eval> (define Describe-Verwahrkonto
        (lambda (K)
          (cond ((Verwahrkonto? K)
                 (Describe-Konto K)
                 (newline)
```

```
                  (display "Buchungsgrund:   ")
                  (display
                   (Verwahrkonto-Buchungsgrund K))
                  (newline)
                  (display "Bearbeiter:      ")
                  (display
                   (Verwahrkonto-Bearbeiter K)))
                 (#t (error
                      "Kein Verwahrkonto: " K)))))
eval> (Describe-Verwahrkonto V001) ==>

______ Inhalt des Kontos: 3 ______
Name-Inhaber:    Meyer
Adresse-Inhaber: D-21399 Lüneburg
Einzahlungen:    0
Auszahlungen:    0
Kontostand:      0
Buchungsgrund:   Unbekannt
Bearbeiter:      Krause
```

Das skizzierte `define-struct`-Konstrukt ermöglicht eine einfache Vererbung. Es verlangt, dass eine Datenstruktur (Klasse) nur von einer (anderen) Datenstruktur erben kann. Damit hat jede Klasse nur eine Oberklasse (auch Superklasse genannt). Erst mit dem Zulassen mehrerer Oberklassen (multiple Vererbung) gewinnt man Flexibilität, um spezielle Operationen schrittweise (inkrementell) aus Allgemeineren konstruieren zu können. Vorteilhaft wäre auch der Sonderfall einer selbstbezüglichen Vererbung. Eine bestehende Struktur könnte ohne großen Aufwand um einige Felder erweitert werden, wenn die Ergänzung in die betroffenen Objekte (Instanzen) übernommen („propagiert“) würde.

```
eval> (define-struct Klasse-I
        (slot-I slot-II) #:mutable #:transparent)
eval> (define Foo
        (make-Klasse-I "value-I" "value-II"))
eval> Foo ==>
      #(struct:Klasse-I "value-I" "value-II")
eval> (define-struct (Klasse-I Klasse-I)
        (slot-III) #:mutable #:transparent)
eval> (define Baz
        (make-Klasse-I
          "value-1" "value-2" "value-3"))
eval> Baz ==>
      #(struct:Klasse-I "value-1" "value-2"
                                  "value-3")
eval> (Klasse-I-slot-III Baz) ==> "value-3"
eval> (Klasse-I-slot-III Foo) ==> ERROR ...
      ;Klasse-I-slot-III: expects args of type
      ; <struct:Klasse-I>; given instance
      ; of a different <struct:Klasse-I>
```

```
      ; Kein Propagieren der Änderung!
eval> (Klasse-I-slot-I Foo)  ==> "value-I"
```

Für das Propagieren einer solchen Änderung bedarf es der dynamischen Vererbung.

2.6.4 Zusammenfassung: Zugriffskonstrukte als Schnittstelle

Das `define-struct`-Konstrukt generiert den Verbund von Konstruktor, Selektor, Prädikat und Mutator. Der Konstruktor `make-`$<$ *structure*$-$*name* $>$ bildet ein Objekt mit den Feldern (engl.: *slots*) und kann diese mit voreingestellten Werten (engl.: *default values*) initialisiert.

Ein generierter Selektor besteht aus dem Strukturnamen und dem Feldnamen, verbunden durch einen Bindestrich. Das Prädikat hat als Namen den Strukturnamen ergänzt um ein Fragezeichen am Namensende. Ein Feldinhalt einer mit `make-`$<$ *structure*$-$*name* $>$ gebildeten Struktur ist mit dem `set-`$<$ *structure*$-$*name* $>$`-`$<$ *slot* $>$`!`-Konstrukt modifizierbar.

Das `define-struct`-Konstrukt ermöglicht eine einfache statische Vererbung. Eine Struktur kann auf einer vorher definierten Struktur aufbauen.

Charakteristisches Beispiel für Abschnitt 2.6

Das folgende Konstrukt `Typbestimmung` basiert auf einem einfachen Bestimmungsbaum (engl.: *discrimination tree*).[16] Für diesen Baum definieren wir zwei Strukturen, eine für Knoten und eine weitere für Baumblätter. Beide mit `define-struct`-Konstrukten definierten Strukturen müssen auf „*top level*“ erklärt werden und können daher nicht in das `letrec`-Konstrukt von `Typbestimmung` integriert werden.

```
eval> (define-struct Baum-Blatt
        (Wert) #:mutable #:transparent)

eval> (define-struct Knoten
        (Test Ja Nein) #:mutable #:transparent)

eval> (define Typbestimmung
        (lambda (Sexpr)
          (letrec
           ((Naechster-Knoten
             (lambda (Testergebnis Baum)
               (if Testergebnis
                   (Knoten-Ja Baum)
                   (Knoten-Nein Baum))))
```

[16] Näheres zur Konstruktion von „*discrimination nets*“ ↪ z. B. [36].

```
            (Bestimmung
             (lambda (Sexpr Baum)
               (cond ((Baum-Blatt? Baum)
                         (Baum-Blatt-Wert Baum))
                     ((Knoten? Baum)
                      (Bestimmung
                       Sexpr
                       (Naechster-Knoten
                        ((Knoten-Test Baum) Sexpr)
                        Baum)))
                     (#t #f))))
            (LISP-Typen
             (make-Knoten
              vector?
              (make-Baum-Blatt "Vektor")
              (make-Knoten
               pair?
               (make-Baum-Blatt "Paar")
               (make-Knoten
                symbol?
                (make-Baum-Blatt "Symbol")
                (make-Knoten
                 number?
                 (make-Baum-Blatt "Zahl")
                 (make-Baum-Blatt
                  "Unbekannt")))))))
          (Bestimmung Sexpr LISP-Typen))))

eval> (Typbestimmung 'Foo)
      ==> "Symbol"
eval> (Typbestimmung "OK!")
      ==> "Unbekannt"
eval> (Typbestimmung (list 1))
      ==> "Paar"
```

2.7 Module (Lokaler Namensraum)

Ein Namensraum (≈ Umgebung) stellt Symbol-Wert-Assoziationen bereit (↪ Abschnitt 12 S. 42). Der gleiche Name, d. h. das gleiche Symbol, kann in unterschiedlichen Umgebungen zu verschiedenen Werten führen, wie mittels Programm 2.8 S. 298 im vorhergehenden Abschnitt ausführlich gezeigt wurde. Der Namensraum ist ein zweckmäßiges Konstruktionsmittel um Namenskonflikten vorzubeugen. Werden fremde Konstrukte importiert oder arbeiten mehrere Programmierer gemeinsam an einer Problemlösung, dann besteht die Gefahr, dass für verschiedene Konstrukte die gleichen Namen verwendet werden. Das Einbetten jeder Teillösung in einen eigenen Namensraum verhindert das ungewollte

Überschreiben einer Symbol-Wert-Assoziation.

Der Namensraum übernimmt eine Abschottungsaufgabe. Er „verpakkt“ eine (Teil)Lösung zu einem *Paket*. Ein Paket, auch als *Modul* bezeichnet, stellt somit eine Sammlung von Symbol-Wert-Assoziationen bereit. Der Paketname ist ein Abstraktionsmittel. Er ermöglicht, diese Sammlung als eine „Einheit“ zu benutzen.

Module sind mit Modulen kombinierbar. Sie können aus schon existierenden Modulen konstruiert werden. Dazu kann ein Modul mit anderen Modulen „kommunizieren“. Diese Kommunikation bedingt, dass Module über gemeinsame Namen verfügen. Wir können drei Arten von Namen unterscheiden:

1. *Importnamen*,
   Dies sind Namen, die in anderen Moduln definert sind und im vorliegenden Modul genutzt werden.

2. *Exportnamen* und
   Dies sind Namen, die, falls Bedarf besteht, von anderen Moduln genutzt werden können.

3. *Interne Namen*
   Dies sind Namen, nur für den internen Gebrauch im vorliegenden Modul.

2.7.1 Import- und Exportnamen

Die Import- und Exportnamen bilden die Schnittstelle des Moduls. Sie sind die Grundlage um die Modularisierung einer komplexen Aufgabe zu spezifizieren (Näheres ↪ Abschnitt 3.2.2 S. 411).

Module können in einer Hierarchie angeordnet werden. Entsprechend dieser Struktur sind ihre Namen verfügbar. Nicht zugreifbar sind Namen für Module auf höherer Ebene, während umgekehrt der Zugriff von tieferen Ebenen entlang der Hierarchie nach oben erfolgen kann. Module bilden damit sogenannte Blockstrukturen, wie sie erstmals in *ALGOL* (Akronym für *ALGOrithmic Language*) realisiert wurden. Anhand des Beispiels Kontenverwaltung erörtern wir das „Verpacken“ von Symbolen mittels des *PLT-Scheme (DrScheme)* Modul-Konzeptes.

Zur Einführung in das Modul-Konzept betrachten wir vorab das einfache Beispiel „`module Termin`“. Das `require`-Konstrukt dient zum Importieren, das `provide`-Konstrukt kennzeichnet das zum Import Bereitgestellte.

```
eval> (module Termin (lib "scheme")
        (require (lib "scheme/date.ss"))
        (provide heute date->string)
         (define heute
           (string-append
```

```
            "Heute: "
            (date->string
            (seconds->date
              (current-seconds))))))
```

Bevor der Wert des Exportsymbols `heute` abgefragt werden kann, ist unser `module Termin` zu importieren; hier in unsere *top-level*-Umgebung, und zwar mit dem `require`-Konstrukt.

```
eval> heute ==> ERROR ...
eval> (require 'Termin)
eval> heute ==> "Heute: Monday, January 4th, 2010"
eval> date->string
      ==> #<procedure:date->string>
```

Der Namensraum eines Moduls wird besonders geschützt und vom Namensraum der normalen Definitionen unterschieden. Z. B. ist der Mutator `set!` auf `heute` daher nicht direkt anwendbar.

```
eval> (set! heute 20100104) ==> ERROR ...
      ;set!: cannot mutate module-required
      ; identifier in: heute
```

Die Übernahme von Namen, die mit dem `provide`-Konstrukt bereitgestellt werden, lässt sich im `require`-Konstrukt einschränken und zwar mit `only-in` und `except-in` wie das folgende Beispiel zeigt:

```
eval> (module Foo scheme
        (provide get-A get-B)
        (define A 'a)
        (define B 'b)
        (define get-A
          (lambda () A))
        (define get-B
          (lambda () B)))
eval> (module Bar scheme
        (require (only-in 'Foo get-B))
        (cons (get-B) (get-B)))
eval> (require 'Bar) ==> (b . b)
eval> (require (except-in 'Foo get-B))
eval> get-A ==> #<procedure:get-A>
eval> get-B ==> ERROR ...
                ;reference to an
                ; identifier before
                ; its definition:
                ; get-B
```

Für unsere Kontenverwaltung trennen wir das Management für die Konten von der „Datenbank“, die die Menge der Konten enthält. Dazu definieren wir die beiden Module `Konten-Datenbank` und `Konten--Management`, wobei der letztgenannte auf dem erstgenannten Modul aufbaut. Daher enthält der Modul `Konten-Management` ein entsprechendes `require`-Konstrukt:

```
eval> (module Konten-Datenbank scheme
        (provide *KONTEN* Get-Konto Add-Konto!)
        (define *KONTEN* ...)
        (define Get-Konto ...)
        (define Add-Konto! ...))
eval> (module Konten-Management scheme
        (provide ...)
        (require 'Konten-Datenbank)
        (define-struct Konto ...))
```

Um das Beispiel leicht verstehbar zu halten, werden die Konten als einfache Assoziationsliste (A-Liste ↪ Abschnitt 2.2.2 S. 183) in dem Modul `Konten-Datenbank` an das Symbol `*KONTEN*` gebunden. Außerdem sind nur die beiden Funktionen `Get-Konto` und `Add-Konto!` ohne Prüfung ihrer Argumente implementiert.

```
eval> (module Konten-Datenbank scheme
        (provide *KONTEN*
                 Get-Konto
                 Add-Konto!)
        (define *KONTEN* (list))
        (define Get-Konto
          (lambda (Konto)
            (let ((k (assoc Konto
                            *KONTEN*)))
              (cond ((pair? k) (car k))
                    (#t k)))))
        (define Add-Konto!
          (lambda (Konto)
            (set! *KONTEN*
                  (cons (list Konto)
                        *KONTEN*)))))
```

Für die Beschreibung des Kontos nutzten wir das `define-struct`-Konstrukt (↪ Abschnitt 2.6.2 S. 309). Die damit verbundenen Konstrukte, wie z. B. `make-Konto`, `Konto?`, `set-Konto-Kontostand` etc., sind über das `provide`-Konstrukt bereitzustellen. Die Konstrukte `Get--Konto` und `Add-Konto!` von `Konten-Datenbank` sind „durchzureichen“, also auch im `provide`-Konstrukt zu nennen.

```
eval> (module Konten-Management scheme
        (provide
             make-Konto
             Konto?
             Konto-Identifizierung
             Konto-Name-Inhaber
             Konto-Adresse-Inhaber
             Konto-Kontostand
             set-Konto-Kontostand!
             Get-Konto
```

```
          Add-Konto!)
    (require 'Konten-Datenbank)
    (define-struct Konto
      (Identifizierung
       Name-Inhaber
       Adresse-Inhaber
       (Einzahlungen #:auto)
       (Auszahlungen #:auto)
       (Kontostand #:auto))
       #:transparent
       #:mutable
       #:auto-value 0))
```

Auf die *top-level*-Ebene importiern wir den Modul `Konten-Management` mit dem `require`-Konstrukt und erzeugen dann drei Konten:

```
eval> (require 'Konten-Management)
eval> (define K1 (make-Konto 1
                             "Leuphana Universität"
                             "Lüneburg"))
eval> (define K2 (make-Konto 2
                             "Siemens AG"
                             "München"))
eval> (Add-Konto! K1)
eval> (Add-Konto! K2)
eval> (Konto-Name-Inhaber (Get-Konto K1))
      ==> "Leuphana Universität"
eval> (define K3 (make-Konto 3
                             "Software AG"
                             "Darmstadt"))
eval> (Get-Konto K3) ==> #f
```

Die Konten, gebunden an das Symbol `*KONTEN*`, sind auf der *top-level*-Ebene nicht (direkt) verfügbar, weil sie in einem anderen Namensraum gebunden sind. Über die Konstrukte `Add-Konto!` und `Get-Konto` können wir aber darauf zugreifen.

```
eval> *KONTEN* ==> ERROR ...
      ;reference to an identifier
      ; before its definition: *KONTEN*
eval> (define K3 (make-Konto 3
                             "Software AG"
                             "Darmstadt"))
eval> (Get-Konto K3) ==> #f
eval> (Add-Konto! K3)
eval> (Konto-Adresse-Inhaber (Get-Konto K3))
      ==> "Darmstadt"
```

Mit dem `require`-Konstrukt können wir jedoch die `Konten-Datenbank`, also `*KONTEN*`, auch auf *top-level*-Ebene importieren.

```
eval> (require 'Konten-Datenbank)
eval> *KONTEN* ==>
((#(struct:Konto 3 "Software AG" "Darmstadt" 0 0 0))
 (#(struct:Konto 2 "Siemens AG" "München" 0 0 0))
 (#(struct:Konto 1 "Leuphana Universität" "Lüneburg"
                                          0 0 0)))
```

Allerdings ist das Symbol `*KONTEN*` vor Veränderungen auf der *top-level*-Ebene geschützt:

```
eval> (set! *KONTEN* 7) ==> ERROR ...
      ;set!: cannot mutate module-required
      ; identifier in: *KONTEN*
```

Mit der Kenntnis des Namensraums können wir jedoch Ausdrücke im Namensraum evaluieren, so dass wir die Sperre „austricksen“ könnten:

```
eval> (eval '(set! *KONTEN*
               (cons (list 'A) *KONTEN*))
                 (module->namespace
                   ''Konten-Datenbank))
eval> *KONTEN* ==>
((A)
 (#(struct:Konto 3 "Software AG" "Darmstadt" 0 0 0))
 (#(struct:Konto 2 "Siemens AG" "München" 0 0 0))
 (#(struct:Konto 1 "Leuphana Universität" "Lüneburg"
                                           0 0 0)))
```

Klar ist, eine solche Durchbrechungung des „Modul-Schutzraumes“ wäre keine zweckmäßige Maßnahme. Wir nutzen Module ja gerade wegen ihres eigenen Namensraumes.

Exkurs: *PLaneT*

PLaneT ist das zentrale System von *PLT-Scheme* zur Distribution von Moduln (Paketen):
↪ `http://planet.plt-scheme.org/` (Zugriff: 15-Jan-2010).
Einige Module können vom *PLaneT*-Server automatisch heruntergeladen werden. Z. B. wenn man das erste Mal folgendes `require`-Konstrukt evaluiert.

```
eval> ;;;Automatisches Download von Version 1.0
      ;;; der "random.plt" Bibliothek
      ;;; und dann importieren des
      ;;; Moduls "random.ss".
      (require
        (planet "random.ss"
          ("schematics" "random.plt" 1 0)))
eval> (random-gaussian)
      ==> 0.7386912134436788
```

Das `planet`-Konstrukt ermöglicht die Auswahl anhand von Versionsnummern (Näheres dazu ↪ *PLT-Scheme* Online-Dokumentation).

2.7.2 Zusammenfassung: Module

Module sind mit Modulen kombinierbar und können aus schon existierenden Modulen konstruiert werden. Ihre Kommunikation erfolgt über eine Import- (`require`) und eine Exportschnittstelle (`provide`). Interne Namen sind nicht importierte und nicht als Export bereitgestellte Namen.

Module übernehmen eine „Abschottungsaufgabe“. Daher sind die internen Namen vor dem Zugriff von außen, d. h. von anderen Moduln aus, geschützt.

Charakteristisches Beispiel für Abschnitt 2.7

Die Datei `Selbstreplikation.ss` enthält den gleichnamigen Modul, der eine zufällige Folge aus kleinen Buchstaben erzeugt und diese auf dem Bildschirm ausgibt. Dabei wird ein Evolutionsprozess bezüglich dieser Buchstabenfolge simuliert und zwar in Form der Modifikation des ausgebenden `lambda`-Konstruktes. Die Buchstaben sind 4 Häufigkeitsgruppen zugeordnet. Ihre Einteilung entspricht dem *Morsecode*.[17]

Bei den Standardeinstellungen von *PLT-Scheme*, Version 4.2.3, speichern wir den Modul `Selbstreplikation` in folgendem Pfad:

```
C:\Dokumente und Einstellungen\bonin\Anwendungsdaten\
     PLTScheme\4.2.3\collects\HEGB\Selbstreplikation.ss
```

Die Datei `Selbstreplikation.ss` hat folgenden Inhalt:

```
;;;;Erzeugung von Buchstabenketten
;;;; mittels Zufallsgenerator
(module Selbstreplikation scheme
  (provide Evolution)
;;;Ausgang der Evolution ist das Konstrukt Kernel
;;; Häufigkeit der Buchstaben nach Morse in 4 Gruppen
;;; 4 wertig: e, t
;;; 3 wertig: a,i,m,n
;;; 2 wertig: d,g,k,o,r,s,u,v
;;; 1 wertig: b,c,f,h,j,i,p,q,v,x,y,z
;;;
;;;Ende wenn Kernel länger 80 Zeichen
  (define Kernel '(lambda () (display "---\n")))
  (define Offset
    (lambda ()
      (let* ((Mutation (random 99))
             (Rest1 (cdr Kernel))
             (Rest2 (cdr Rest1))
             (Rest3 (cdr Rest2))
             (Rest4
```

[17] Der *Morsecode*, auch als *Morsealphabet* bezeichnet, wurde *Samuel Morse* im Jahr 1833 entwickelt.

```
             (cond
              ((= (length Rest3) 0) Rest3)
              ((pair? Rest3) (cdr Rest3))
              (#t (error "Kein Pair!"))))
            (set-Kernel!
             (lambda (Buchstabe)
               (set! Kernel
                 (append
                  Kernel
                  `((display ,Buchstabe)))))))
       (cond
        ((> (length Kernel) 80) (error "Ende!"))
        ((< Mutation 3) (set-Kernel! "a"))
        ((< Mutation 4) (set-Kernel! "b"))
        ((< Mutation 5) (set-Kernel! "c"))
        ((< Mutation 7) (set-Kernel! "d"))
        ((< Mutation 11) (set-Kernel! "e"))
        ((< Mutation 12) (set-Kernel! "f"))
        ((< Mutation 14) (set-Kernel! "g"))
        ((< Mutation 15) (set-Kernel! "h"))
        ((< Mutation 18) (set-Kernel! "i"))
        ((< Mutation 19) (set-Kernel! "j"))
        ((< Mutation 21) (set-Kernel! "k"))
        ((< Mutation 22) (set-Kernel! "l"))
        ((< Mutation 25) (set-Kernel! "m"))
        ((< Mutation 28) (set-Kernel! "n"))
        ((< Mutation 30) (set-Kernel! "o"))
        ((< Mutation 31) (set-Kernel! "p"))
        ((< Mutation 32) (set-Kernel! "q"))
        ((< Mutation 33) (set-Kernel! "r"))
        ((< Mutation 35) (set-Kernel! "s"))
        ((< Mutation 39) (set-Kernel! "t"))
        ((< Mutation 41) (set-Kernel! "u"))
        ((< Mutation 42) (set-Kernel! "v"))
        ((< Mutation 44) (set-Kernel! "w"))
        ((< Mutation 45) (set-Kernel! "x"))
        ((< Mutation 46) (set-Kernel! "y"))
        ((< Mutation 47) (set-Kernel! "z"))
        ((< Mutation 50) (set-Kernel! " "))
        (#t
         (set! Kernel
           (cons
            (car Kernel)
            (cons
             (car Rest1)
             (cons (car Rest2) Rest4)))))))))
(define Evolution
  (lambda ()
    ((eval Kernel))
```

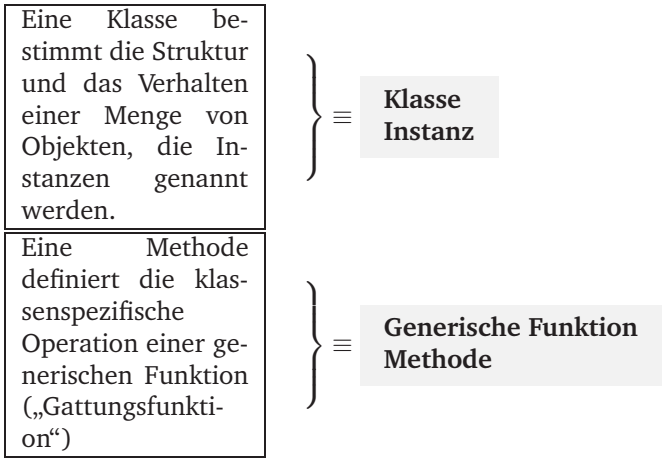


Abbildung 2.22: *CLOS*-Bausteine — Objekttypen —

```
      (Offset)
      (Evolution))))
```

Auf der *top-level*-Ebene importieren wir den Modul und wenden das Konstrukt `Evolution` wie folgt an:

```
(require HEGB/Selbstreplikation)
(Evolution) ==> ...
   ddy junmmmotiukyceeai  ...
   ...\collects\HEGB\Selbstreplikation.ss:22:33: Ende!
```

2.8 Klasse-Instanz-Modell

Die Realisierung von *Klasse-Instanz-Modellen* hat in LISP eine lange Tradition. Als charakteristische Vertreter gelten die beiden Systeme *Common LOOPS* und *Flavors*:

- *Common LOOPS* (*Common LISP Object-Oriented Programming System*, ↪ [19])

  Common LOOPS, implementiert in *Common LISP*, realisiert die Objekt-Orientierung mit generischen Funktionen, so dass normale LISP-Funktionen und Methoden syntaktisch einheitlich gehandhabt werden können. Meta-Objekte sind eine charakteristische *Common LOOPS*-Eigenschaft.

- *(New) Flavors* (↪ [131, 188])

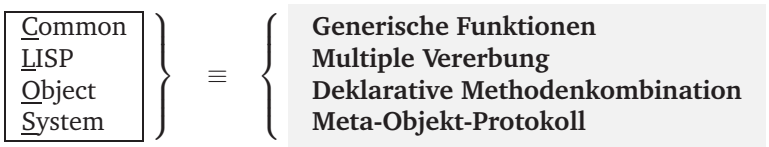


Abbildung 2.23: *CLOS*-Bausteine —- Charakteristika —

> Das *Flavor*-System wurde ursprünglich für die LISP-Maschine entwickelt (↪ Abschnitt A S. 457). Im Mittelpunkt steht ein *Mixen* von Teilkomponenten durch multiple Vererbung. Dieser „*mixin style*“ begründet den außergewöhnlichen Namen *Flavor*. Durch das Mischen von verschiedenen *Flavors* erhält man einen neuen *Flavor*. Dabei schmeckt man beim Zusammenmischen die einzelnen Komponenten bis hin zur Systemwurzel „*Vanilla-flavor*“.

Selbst kleine LISP-Implementationen ermöglichen den Umgang mit Klassen und Instanzen. Ohne großen Aufwand kann ein LISP-System für eine objektgeprägte Programmierung erweitert werden (↪ z. B. [53, 9, 25, 107, 62]). Entstanden ist daher eine Vielfalt von LISP-Erweiterungen. Sie weisen wesentliche Unterschiede in ihren Konstrukten und in ihrer Terminologie auf. Begriffe wie Objekt, Klasse, Instanz, Methode oder Funktion werden unterschiedlich gehandhabt. Diese Sprach- und Konstruktevielfalt war Veranlassung einen leistungsfähigen Standard zu schaffen. Die amerikanischen Standardisierungsbemühungen haben zum *Common LISP Object System* (kurz: *CLOS*) geführt (↪ *ANSI X3J13* und [20]). In Abbildung 2.22 S. 327 sind die *CLOS*-Bausteine genannt. Die *CLOS*-Charakteristika zeigt Abbildung 2.23 S. 328.

Mit dem `define-struct`-Konstrukt ist eine „Klasse“ (Struktur) auf der Basis einer existierenden Klasse definierbar (↪ Abschnitt 2.6.3 314). Vererbt werden dabei die Slots einer Klasse. Ein wesentliches Leistungsmerkmal eines Klassen-Instanz-Modells sind seine Optionen der Vererbung, d. h. die Möglichkeit neue Klassen durch Rückgriff auf die Eigenschaften von mehreren Klassen zu definieren (↪ Abbildung 2.23 S. 328). Ein weiteres Beurteilungskriterium ist der Zeitpunkt der Vererbung. Hierbei ist zwischen statischer und dynamischer Vererbung zu unterscheiden.

Einige der vielfältigen Optionen der Vererbung verdeutlichen wir anhand der Erweiterung unserer Kontenverwaltung (↪ Abschnitte 2.7 319). Die Klasse `Verwahrkonto` erbt zusätzlich zu den Eigenschaften der Klasse `Konto` die Eigenschaften einer Klasse `Offener-Vorgang`. Die Klasse `Offener-Vorgang` definiert z. B. für die Klasse `Verwahrkonto` das bisher nicht vorhandene Feld `Aktenzeichen`.

2.8.1 Object-Oriented Programming System

Im Folgenden nutzen wir zunächst Optionen des `class`-Konstruktes von *PLT-Scheme*[18] mit der Entwicklungsumgebung von *DrScheme* (Version 4.2.3; Dez. 2009). Den Zugriff auf die Slots einer Instanz, d. h. auf ihren lokalen Wert, erfolgt mit Hilfe des `send`-Konstruktes. Die Selektoren und Mutatoren sind an das jeweilige Objekt (Instanz oder Klasse) im Sinne einer Nachricht zu „senden“.

In der funktions-geprägten Terminologie formuliert: Das `send`-Konstrukt hat als Argumente das Objekt, den Operationsnamen und gegebenenfalls Argumente für die Operation. Weil wir einen Selektor oder Mutator mit dem `send`-Konstrukt an ein Objekt „übermittelt“, betrachten wir diese in der objekt-geprägten Terminologie als eine „Nachricht“ (engl.: *message*), die das Objekt empfängt und verarbeitet.

Aus der Kommunikationsicht (Nachrichtenübermittlung) können wir formulieren: Empfängt ein Objekt eine Nachricht, dann interpretiert es die Nachricht, d. h. es ermittelt die zu applizierenden (Teil-)Operationen und führt diese aus. Eine Nachricht kann mehrere (Teil-)Operationen betreffen. Wir sprechen in diesem Kontext eher von Methoden (engl.: *methods*) statt von Operationen oder Funktionen. Eine Methode ist eine Funktion, die als Reaktion auf eine zugehörende Nachricht ausgeführt wird. Sie ist einer Klasse zugeordnet. Die Vererbung bezieht sich daher sowohl auf die lokalen Zustandsbeschreibungen (Slots) wie auf die generierten Selektions-/Mutations-Funktionen und die selbst definierten Funktionen (Methoden).

Für unsere Kontenverwaltung definieren wir als Ausgangsbasis zunächst die Klasse `Konto`. Entsprechend der `define-struct`-Lösung (↪ Abschnitt 2.6.2 S. 309) beschreibt die Klasse `Konto` die *Slots* (Felder): `Idenifizierung`, `Name-Inhaber`, `Adresse-Inhaber`, `Einzahlungen`, `Auszahlungen` und `Kontostand`.

Hinweis: Name einer Klasse.
In üblicher LISP-ischer Terminologie endet ein Klassenname mit einem Prozentzeichen, so dass unser Konto jetzt `Konto%` genannt wird.

`Konto%`-Konstrukt

```
eval> (define Konto%
        (class object%
          ;Argument zur
          ; Initialisierung
          (init id name)
          ;
          ;Slots
          ; public
```

[18] *PLT-Scheme* ↪`http://www.plt-scheme.org/` (Zugriff: 24-Oct-2009)

```
(field (Identifizierung id))
(field (Name-Inhaber name))
(field (Adresse-Inhaber ""))
; private
(define Einzahlungen null)
(define Auszahlungen null)
(define Kontostand 0)
;
;Initialisierung der
;Superklasse
(super-new)
;
;Methoden:
; Selektoren
(define/public get-Identifizierung
  (lambda () Identifizierung))
(define/public get-Name-Inhaber
  (lambda () Name-Inhaber))
(define/public get-Adresse-Inhaber
  (lambda () Adresse-Inhaber))
(define/public get-Einzahlungen
  (lambda () Einzahlungen))
(define/public get-Auszahlungen
  (lambda () Auszahlungen))
(define/public get-Kontostand
  (lambda () Kontostand))
;Methoden:
; Mutatoren
(define/public set-Name-Inhaber!
  (lambda (Name)
    (set! Name-Inhaber Name)))
(define/public set-Adresse-Inhaber!
  (lambda (Adresse)
    (set! Adresse-Inhaber Adresse)))
(define/public set-Einzahlungen!
  (lambda (Wert)
    (set! Einzahlungen Wert)))
(define/public set-Auszahlungen!
  (lambda (Wert)
    (set! Auszahlungen Wert)))
;Nur intern zugreifbar
(define set-Kontostand!
  (lambda (Wert)
    (set! Kontostand Wert)))))
```

Konstruktor: Für das Kreieren eines Kontos ist das `new`-Konstrukt zu applizieren. Sein erstes Argument ist die Klasse, hier `Konto%`. Da das `new`-Konstrukt ein `class`-Konstrukt bedingt, verteilt sich die originäre Konstruktor-Aufgabe (Definition und Implementation) auf beide Konstrukte. Als Kontenbeispiele konstruieren wir `A001` und `A002`, wie folgt:

```
eval> (define A001
```

```
        (new Konto% (id 1)
                    (name "Hinrich Bonin")))
eval> (define A002
        (new Konto% (id 2)))
      ==> ERROR ...
     ;instantiate: no argument for required
     ; init variable: name in class: Konto%
eval> (define A002
        (new Konto% (id 2) (name "unbekannt")))
eval> A001 ==> #(struct:object:Konto% ...)
eval> (send A001 get-Name-Inhaber)
      ==> "Hinrich Bonin"
```

Prädikat: Das `class?`-Konstrukt erkennt eine Klasse. Ob es sich um unsere Klasse `Konto%` oder um eine andere Klasse handelt, kann es nicht feststellen. Zur Prüfung auf unsere Klasse `Konto%` definieren wir mittels der Umwandelung eines `struct`-Objektes in ein `vector`-Objekt ein eigenes Prädikat `Konto%?`.

```
eval> (define-syntax-rule (Konto%? x)
        (and (class? x)
             (eq? 'struct:class:Konto%
                  (vector-ref (struct->vector x) 0))))
eval> (Konto%? Konto%) ==> #t

eval> (define Foo%
        (class object% (super-new)
          (define i 7)))
eval> (Konto%? Foo%) ==> #f
```

Neben der Feststellung, ob unsere Klasse `Konto%` vorliegt, benötigen wir ein Prädikat zur Erkennung einer Instanz dieser Klasse `Konto%`. Eine Instanz erkennt das Prädikat `object?`. Auf dieser Basis definieren wir analog zum Konstukt `Konto%?` das Konstrukt `Konto?`.

```
eval> (object? A001) ==> #t
eval> (define-syntax-rule (Konto? x)
       (and (object? x)
            (eq? 'struct:object:Konto%
                 (vector-ref (struct->vector x) 0))))
eval> (Konto? A001) ==> #t
eval> (Konto? (new Foo%)) ==> #f
```

Bei den beiden Prädikaten `Konto%?` und `Konto?` nutzen wir die jeweilige Kennzeichnung `struct:class:Konto%` und `struct:object:Konto%` und greifen damit auf Details der Implementation zurück.

Für unsere Kontenverwaltung benötigen wir, wie im Fall der `define-struct`-Lösung (↪ Abschnitt 2.6.1 S. 306) wieder die Prädikate `Betrag?` und `Buchungsdatum?`.

```
eval> (define Betrag?
        (lambda (X)
          (and (number? X) (>= X 0))))
eval> (define Buchungsdatum?
        (lambda (X)
          (and (number? X) (>= X 100101))))
```

Selektor: Die mit `define/public` deklarierten Methoden in einer Klasse, hier `Konto%`, werden von ausserhalb der Klasse mit dem `send`-Konstrukt unter Angabe einer Instanz, hier z. B. `A001`, dieser Klasse appliziert, und zwar so wie ein Scheme übliches `lambda`-Konstrukt.

`eval> (send` $< object-expr >$ $< method-id >$ $\{ < argument > \}$`)`

mit:

$< object-expr >$	$\equiv$ Evaluiert zu einem Objekt.
$< method-id >$	$\equiv$ (Externer) Name der Mehode.
$< argument >$	$\equiv$ Argument für die Applikation der Methode, falls erforderlich.

Damit kann z. B. die Methode `get-Name-Inhaber` wie folgt selektiert und angewendet werden:

```
eval> (define A001
        (new Konto% (id 1)
                    (name "Hinrich Bonin")))
eval> (send A001 get-Name-Inhaber)
      ==> "Hinrich Bonin"
```

Auch auf einen Slot lässt sich von ausserhalb der Klasse zugreifen, wenn er nicht als *private field* mit dem `define`-Konstrukt deklariert wurde. Dazu dient das `get-field`-Konstrukt:

`eval> (get-field` $< field-id >$ $< object-expr >$`)`

mit:

$< field-id >$	$\equiv$ Name des Feldes (Slots).
$< object-expr >$	$\equiv$ Evaluiert zu einem Objekt.

Wir können z. B. somit auch direkt auf den Slot `Name-Inhaber` zugreifen:

```
eval> (get-field Name-Inhaber A001)
      ==> "Hinrich Bonin"
```

Im Sinne der Denkwelt der Objekt-Orientierung ist ein Zugriff über den Methodenaufruf angebrachter, also daher:

```
eval> (send A001 get-Name-Inhaber)
      ==> "Hinrich Bonin"
```

Exkurs: Begriff *Messagepassing*
Im strengen Sinne des Nachrichtenversands verursacht das Senden einer Nachricht an jemanden ein (potentiell) asynchrones Anworten auf diese Nachricht. Beispiel: Der Sachbearbeiter sendet seinem Chef eine Nachricht. Wann er die Antwort erhält, bestimmt in der Regel der Chef. Zwischenzeitlich kann der Sachbearbeiter andere Aufgaben erledigen. Die `send`-Syntax suggeriert, dass ein solches „asynchrones" Antwortverhalten vorliegt. In objekt-orientierten LISP-Erweiterungen bedeutet Nachrichtenversand das Heraussuchen einer Funktion und ihre Abarbeitung. Der Sender ist nicht in der Lage, zwischenzeitlich etwas anderes zu bearbeiten. Er wartet auf die Antwort. Senden einer Nachricht führt nur zur Applikation einer LISP-Funktion.

Das Klasse-Instanz-Modell hat als ein wesentliches Merkmal die Abschirmung („Einkapselung") der internen Zustände ihrer Objekte (Klassen und Instanzen) gegen Zugriffe von aussen, die nicht über die vorgesehene Schnittstelle laufen.

Exkurs: Begriff *Encapsulation*
Der gewünschte Objektzustand ist entscheidend für die Objektdefinition und für die Objektverwendung. Die Einkapselung (engl.: *encapsulation*) hat deshalb nur die Implementation zu verbergen und nicht den internen Objektzustand. Die Zusammenfassung aller Implementationsinformationen zu einem Objekt (Programmstück) und deren Abschirmung gegen direkte Zugriffe von anderen Programmteilen aus, sichert die Modifizierbarkeit eines Objektes. Neben- und Fernwirkungen einer Objektänderung sind dadurch begrenzbar. Wäre jedoch eine Optimierung der Effizienz gefordert, dann hätte eine Streuung der Informationen über die Implementationsdetails an viele Programmstellen Vorteile. Umwege über die Schnittstelle, d. h. hier über den Nachrichtenversandmechanismus, wären einsparbar.

Mutator: Wie Selektoren werden auch Mutatoren generiert, die über das `send`-Konstrukt anwendbar sind.

```
eval> (send A001 set-Name-Inhaber! "Emma Musterfrau")
eval> (send A001 get-Name-Inhaber)
      ==> "Emma Musterfrau"
eval> (send A001 set-Kontostand! 100000.00)
      ==> ERROR ... ; nicht public
          ;send: no such method:
          ; set-Kontostand! for class: Konto%
```

Die anwendungsspezifischen Mutatoren `Add-Einzahlung!` und `Add-Auszahlung!` bauen wir mit den Prädikaten `Buchungsdatum` und `Betrag?` in die Klasse `Konto%` (↪ S. 334) ein. Dabei greifen wir innerhalb der Klasse mittels der Selektoren auf die einzelnen *Slots* zu.

Klasse: `Konto%`

```
eval> (define Konto%
        (class object%
          ;Argument zur
          ; Initialisierung
          (init id name)
          ;
          ;Slots
          ; public
          (field (Identifizierung id))
          (field (Name-Inhaber name))
          (field (Adresse-Inhaber ""))
          ; private
          (define Einzahlungen null)
          (define Auszahlungen null)
          (define Kontostand 0)
          ;
          ;Initialisierung der
          ;Superklasse
          (super-new)
          ;
          ;Methoden:
          ; Selektoren
          (define/public get-Identifizierung
            (lambda () Identifizierung))
          (define/public get-Name-Inhaber
            (lambda () Name-Inhaber))
          (define/public get-Adresse-Inhaber
            (lambda () Adresse-Inhaber))
          (define/public get-Einzahlungen
            (lambda () Einzahlungen))
          (define/public get-Auszahlungen
            (lambda () Auszahlungen))
          (define/public get-Kontostand
            (lambda () Kontostand))
          ;Methoden:
          ; Mutatoren
          (define/public set-Name-Inhaber!
            (lambda (Name)
              (set! Name-Inhaber Name)))
          (define/public set-Adresse-Inhaber!
            (lambda (Adresse)
              (set! Adresse-Inhaber Adresse)))
          (define/public set-Einzahlungen!
            (lambda (Wert)
              (set! Einzahlungen Wert)))
          (define/public set-Auszahlungen!
            (lambda (Wert)
              (set! Auszahlungen Wert)))
          ;Nur intern zugreifbar
          (define set-Kontostand!
            (lambda (Wert)
              (set! Kontostand Wert)))
          ;Anwendungsspezifische
```

```
; Methoden
(define Betrag?
  (lambda (X)
    (and (number? X) (>= X 0))))
(define Buchungsdatum?
  (lambda (X)
    (and (number? X) (>= X 100101))))
(define/public Add-Einzahlung!
  (lambda (E_Datum Betrag)
    (cond ((and (Buchungsdatum? E_Datum)
                (Betrag? Betrag))
           (set-Einzahlungen!
            (cons (list E_Datum Betrag)
                  (get-Einzahlungen)))
           (set-Kontostand!
            (+ Betrag (get-Kontostand))))
          (#t (error
               "Keine Einzahlung hinzugefügt! "
               E_Datum Betrag)))))
(define/public Add-Auszahlung!
  (lambda (A_Datum Betrag)
    (cond ((and (Buchungsdatum? A_Datum)
                (Betrag? Betrag))
           (cond ((<= Betrag (get-Kontostand))
                  (set-Auszahlungen!
                   (cons (list A_Datum Betrag)
                         (get-Auszahlungen)))
                  (set-Kontostand!
                   (- (get-Kontostand) Betrag)))
                 (#t (error
                      "Keine Deckung fuer:"
                      Betrag))))
           (#t (error "Keine Auszahlung möglich! "
                      A_Datum Betrag)))))))
```

Zuerst muss ein neues Konto, hier `A001`, erzeugt werden, damit es die hinzugefügten Methoden kennt. Dann können wir diese Methoden nutzen.

```
eval> (define A001
        (new Konto% (id 1)
                    (name "Hinrich Bonin")))
eval> (send A001 get-Kontostand) ==> 0
eval> (send A001 Add-Einzahlung! 201020 100.00)
eval> (send A001 get-Kontostand) ==> 100.0
eval> (send A001 Add-Auszahlung! 201021 200.00)
      ==> ERROR ... Keine Deckung fuer: 200.0
eval> (send A001 Add-Auszahlung! 201021 45.00)
eval> (send A001 get-Auszahlungen)
      ==> ((201021 45.0))
eval> (send A001 get-Kontostand) ==> 55.0
```

Hinweis: Rückgabewert.
Wird die Methode `Add-Einzahlung!` oder `Add-Auszahlung!` angewendet, dann ist kein Rückgabewert definiert. Falls wir Methoden wie Funktionen schachteln wollten, wäre es zweckmäßig die ganze Instanz zurückzugeben.

```
eval> (send
        ...
        (send
          (send < object > < message_1 >)
          < message_2 > )
        ...
         < message_n >)
```

Notwendig wäre dazu die Kenntnis des Objektes, an das die Nachricht gesendet wurde. Dazu dient hier das Schlüsselwort `this`.

```
eval> (define K% (class object%
                   (field (slot 0))
                   (super-new)
                   (define/public set-Slot!
                     (lambda (Wert)
                       (set! slot (+ slot Wert))
                       this))))

eval> (define Foo (new K%))
eval> (send Foo set-Slot! 4)
      ==> #(struct:object:K% ...)
eval> (send (send Foo set-Slot! 4) set-Slot! 5)
     ==> #(struct:object:K% ...)
eval> (get-field slot Foo) ==> 13
```

Zum Zeitpunkt des Kreierens der Instanz `A002` (↪ S. 330) waren die Methoden `Add-Einzahlung!` und `Add-Auszahlung!` noch nicht definiert. Wir könnten sie trotzdem auf `A002` anwenden, wenn das nachträgliche Modifizieren einer Klasse auf die von ihr abhängigen Objekte unmittelbar vererbt wird. In einem solchen Fall sprechen wir von einer *dynamischen Vererbung*. Als abhängige Objekte gelten alle Klassen und Instanzen, die im Vererbungsgraph auf der modifizierten Klasse aufbauen; oder anders formuliert: alle Objekte die Nachfolgeknoten im Vererbungsgraphen sind. In *PLT-Scheme* [19] mit der Entwicklungsumgebung von *DrScheme* (Version 4.2.3; Dez. 2009) ist die Vererbung nicht dynamisch, so dass wir auf `A002` die später hinzugefügten Methoden nicht anwenden können.

Hinweis: *CLOS: Dynamische Vererbung*.
In *CLOS* ist die Vererbung von Modifikationen, sogar von Instanzvariablen, anwendungsabhängig spezifizierbar. Z. B. kann auf den „alten“ Wert einer

[19] *PLT-Scheme* ↪`http://www.plt-scheme.org/` (Zugriff: 24-Oct-2009)

Instanzvariablen im Rahmen der Vererbung der Modifikation zugegriffen werden (↪ [104]).

2.8.2 Vererbung: *Interfaces*, *Mixins* und *Traits*

Schritt für Schritt zeigen wir anhand der Klassen `Konto%`, `Verwahrkonto%` und `Offener-Vorgang%` einige Möglichkeiten zur Nutzung der Vererbung. Zunächst erbt die Klasse `Verwahrkonto%` von ihrer Oberklasse (*Superclass*) `Konto%` die Eigenschaften, also *Slots*, z. B. `Name-Inhaber` und Methoden, z. B. `get-Name-Inhaber`.

```
eval> (define Verwahrkonto%
        (class Konto%
          (field (Buchungsgrund ""))
          (field (Bearbeiter "unbekannt"))
          (define/public set-Buchungsgrund!
            (lambda (Text)
              (set! Buchungsgrund
                (string-append Text Buchungsgrund))))
          (super-new)))

eval> (define V001
        (new Verwahrkonto%
             (id 7)
             (name "Otto Mustermann")))
eval> (get-field Name-Inhaber V001)
      ==> "Otto Mustermann"
eval> (send V001 get-Name-Inhaber)
      ==>"Otto Mustermann"
```

Die Übergabe der Werte für die Intitialisierung der Klasse `Konto%`, hier `(id 7)` und `(name "Otto Mustermann")`, kann auch ganz oder teilweise in der Klasse `Verwahrkonto%` selbst erfolgen, und zwar durch Nennung im `super-new`-Konstrukt.

```
eval> (define Verwahrkonto%
        (class Konto%
          (init (name "Unbekannt"))
          (field (Buchungsgrund ""))
          (field (Bearbeiter "unbekannt"))
          (define/public set-Buchungsgrund!
            (lambda (Text)
              (set! Buchungsgrund
                (string-append
                  Text Buchungsgrund))))
           ; Gemischte Initialisierung
          (super-new (id 4) (name name))))
eval> (define V001
        (new Verwahrkonto%
              (name "Otto Mustermann")))
```

```
eval> (send V001 get-Name-Inhaber)
      ==> "Otto Mustermann"
```

Innerhalb der Klasse `Verwahrkonto%` wollen wir eine Methode `get--Kurzname-Inhaber` definieren und dabei die Methode `get-Name--Inhaber` der Oberklasse `Konto%` nutzen. Dazu könnte man mit dem Schlüsselwort `this` im `send`-Konstrukt wie folgt formulieren:

```
eval> (define Verwahrkonto%
        (class Konto%
          (field (Buchungsgrund ""))
          (field (Bearbeiter "unbekannt"))
          (define/public set-Buchungsgrund!
            (lambda (Text)
              (set! Buchungsgrund
                (string-append Text Buchungsgrund))))
          (super-new)
          (inherit get-Name-Inhaber)
          (define/public get-Kurzname-Inhaber
            (lambda ()
              (substring
                ;Nachricht an die
                ; ``eigene'' Instanz
                (send this get-Name-Inhaber)
                0 3)))))
eval> (define V001
        (new Verwahrkonto%
             (id 7)
             (name "Otto Mustermann")))
eval> (send V001 get-Kurzname-Inhaber)
      ==> "Ott"
```

Diese Lösung mit `(send this ...)` ist nicht besonders effizient. Alternativ dazu bietet sich eine Lösung mit einem expliziten `inherit`-Konstrukt wie folgt an:

```
eval> (define Verwahrkonto%
        (class Konto%
          (field (Buchungsgrund ""))
          (field (Bearbeiter "unbekannt"))
          (define/public set-Buchungsgrund!
            (lambda (Text)
              (set! Buchungsgrund
                (string-append Text Buchungsgrund))))
          (super-new)
          ;Direkter Zugriff dadurch
          ; möglich
          (inherit get-Name-Inhaber)
          (define/public get-Kurzname-Inhaber
            (lambda ()
```

```
              (substring (get-Name-Inhaber)
                         0 3)))))
eval> (define V001
        (new Verwahrkonto%
             (id 7)
             (name "Otto Mustermann")))
eval> (send V001 get-Kurzname-Inhaber)
      ==> "Ott"
```

Um die Performance ähnlich zum `inherit`-basierten Methodenaufruf zu erzielen, wenn von ausserhalb der Klasse der Methode der Aufruf erfolgt, dann ist das `generic`-Konstrukt anzuwenden.

```
eval> (define get-Verwahrkonto-Name-Inhaber
        (generic Konto% get-Name-Inhaber))
eval> (send-generic V001
        get-Verwahrkonto-Name-Inhaber)
       ==> "Otto Mustermann"
```

Soll eine Methode, deklariert in der vererbenden Klasse (*Superclass*), mit gleichem Namen[20] in der erbenden Klasse (*Subclass*) deklariert werden, und zwar im Sinne des Überschreibens, hier `get-Identifizierung`, dann kann auf die *Superclass*-Methode mittels dem `super`-Konstrukt zugegriffen werden, wie das folgende Beispiel zeigt:

```
eva> (define Verwahrkonto%
        (class Konto%
          (field (Buchungsgrund ""))
          (field (Bearbeiter "unbekannt"))
          (define/public set-Buchungsgrund!
            (lambda (Text)
              (set! Buchungsgrund
                (string-append Text Buchungsgrund))))
          (super-new)
          (inherit get-Name-Inhaber)
          (define/override get-Identifizierung
             (lambda ()
               (string-append
                "Id: "
                (number->string
                 ;Applikation der
                 ; gerade überschriebenen
                 ; Methode
                 (super get-Identifizierung)))))
          (define/public get-Kurzname-Inhaber
             (lambda ()
               (substring (get-Name-Inhaber)
                          0 3)))))
eval> (define V001
```

[20] Präzise: mit gleicher Signatur

```
        (new Verwahrkonto%
             (id 7)
             (name "Otto Mustermann")))
eval> (send V001 get-Identifizierung)
      ==> "Id: 7"
```

Interface

Ein *Interface* (Schnittstelle) wird erklärt, um sicher zu stellen, dass ein Objekt eine vorgegebene Menge von Methoden „versteht“. Das `interface`-Konstrukt hat folgende Syntax:

`eval> (interface (`$< super-interface >$`)` $< id_1 >$ $\{ < id_i > \}$`)`

mit:

$< super-interface >$	$\equiv$ Ein Interface, dessen Deklarationen übernommen werden.
$< id_i >$	$\equiv$ Name der Methode.

Für Verwahrkonten definieren wir ein Interface mit einem sehr vereinfachten Interface ($< super-interface >$) für Konten. Um zu deklarieren, dass die Klasse tatsächlich das angegebene Interface implementiert, ist statt dem `class`-Konstrukt das `class*`-Konstrukt zu verwenden:

```
eval> (define Konto-interface
        (interface ()
          get-Identifizierung))
eval> (define Verwahrkonto-interface
        (interface (Konto-interface)
          set-Buchungsgrund!
          get-Name-Inhaber
          get-Kurzname-Inhaber))
eval> (define Verwahrkonto%
        (class* Konto%
                (Verwahrkonto-interface)
          (field (Buchungsgrund ""))
          (field (Bearbeiter "unbekannt"))
          (define/public set-Buchungsgrund!
            (lambda (Text)
              (set! Buchungsgrund
                   (string-append
                     Text Buchungsgrund))))
          (super-new)
          (inherit get-Name-Inhaber)
          (define/override get-Identifizierung
             (lambda ()
               (string-append
                "Id: "
                (number->string
```

```
                (super get-Identifizierung)))))
          (define/public get-Kurzname-Inhaber
            (lambda ()
              (substring (get-Name-Inhaber) 0 3)))))

eval> (define V001
        (new Verwahrkonto%
             (id 7)
             (name "Otto Mustermann")))
```

Mit dem `is-a?`-Konstrukt, dass als erstes Argument ein Objekt hat und als zweites eine Klasse oder ein Interface, kann festgestellt werden, dass `V001` sowohl dem „Datentyp“ Interface, hier `Verwahrkonto-interface` und `Konto-interface`, wie dem der Klasse, hier `Verwahrkonto%` und `Konto%` entspricht.

```
eval> (is-a? V001 Verwahrkonto-interface)
      ==> #t
eval> (is-a? V001 Verwahrkonto%)
      ==> #t
eval> (is-a? V001 Konto-interface)
      ==> #t
eval> (is-a? V001 Konto%)
      ==> #t
eval> (define Foo
        (class object% (super-new)))
eval> (is-a? V001 Foo)
      ==> #f
```

Mixins

Sogenannte *Mixins* werden mit dem `mixin`-Konstrukt erzeugt. Dabei handelt es sich um eine Funktion, die eine Klassenerweitung erzeugt, wobei die *Superclass* (zunächst) unspezifiziert bleibt. Jedesmal wenn ein *Mixin* für eine spezifizierte Superclass dann appliziert wird, erzeugt es eine neue abgeleitete Klasse, die die entsprechende Erweiterung umfasst. Anhand der Klasse `Offener-Vorgang%` verdeutlichen wir Schritt für Schritt das *Mixins*-Konzept und dabei auch das `mixin`-Konstrukt, das folgende Syntax hat:

$$\texttt{eval> (mixin } (< interface - exprs_c >) \; (< interface - exprs_u >) \; < class - clause_1 > \{ < class - clause_i > \}\texttt{)}$$

mit:

$< interface - exprs_c >$	≡ Die gegebene Klasse muss dieses Interface implementieren.
$< interface - exprs_u >$	≡ Die erzeugte Unterklasse muss dieses Interface implementieren.
$< class - clause_i >$	≡ Die Klauseln, die Erweiterung abbilden, werden beim Evaluieren des `mixin`-Konstruktes auf Übereinstimmung mit den Interfaces hin geprüft.

Wir unterstellen nun, die Analyse für die Entwicklung der Klasse `Offener-Vorgang%` hätte zwei Teilaspekte ergeben, zum Einen ein Thema „Akte“ und zum Anderen ein Thema „Bearbeitung“. Wir formulieren für beide Aspekte entsprechende Slots und Methoden, ohne dabei die Superklasse, hier `Konto%` oder auch `Verwahrkonto%` zu nennen. Die beiden Aspekt formulieren wir allgemein, so dass sie gegebenenfalls auch im Kontext anderer Superklassen genutzt werden könnten.

```
eval> (define Akte-mixin
        (lambda (%)
          (class % (super-new)
            (define Aktenzeichen "XYZ")
            (define Standort "Archiv")
            (define/public get-Aktenzeichen
              (lambda () Aktenzeichen))
            (define/public get-Standort
              (lambda () Standort))
            (define/public set-Aktenzeichen!
              (lambda (Text)
                (set! Aktenzeichen Text)))
            (define/public set-Standort!
              (lambda (Ort)
                (set! Standort Ort))))))

eval> (define Bearbeitung-mixin
        (lambda (%)
          (class % (super-new)
            (field (Aktuelle-Bearbeiter "Fischer"))
            (define Bisheriger-Bearbeiter null)
            (define/public get-Bisheriger-Bearbeiter
              (lambda ()
                Bisheriger-Bearbeiter))
            (define/public set-Bisheriger-Bearbeiter!
              (lambda (Wert)
                (set! Bisheriger-Bearbeiter Wert))))))
```

Jetzt mixen wir beide Aspekte mit unserem `Verwahrkonto%` zur Klasse `Offener-Vorgang%` und erzeugen aus ihr die Instanz `OV001`.

```
eval> (define Offener-Vorgang%
        (Bearbeitung-mixin
          (Akte-mixin Verwahrkonto%)))
eval> (define OV001
```

```
        (new Offener-Vorgang%
          (id 9) (name "Erika Musterfrau")))
eval> (send OV001 set-Bisheriger-Bearbeiter!
                  "Karl Meyer")
eval> (send OV001 get-Bisheriger-Bearbeiter)
       ==> "Karl Meyer"
eval> (get-field Aktuelle-Bearbeiter OV001)
      ==> "Fischer"
eval> (send OV001 get-Identifizierung)
      ==> "Id: 9"
```

Das Mixen von solchen Aspekten („Teillösungen“) birgt mannigfaltige Gefahren, weil wir über die *Superclass* keine Voraussetzungen formuliert haben. Wir könnten z. B. mit einer Klasse `Foo%` mixen:

```
eval> (define Foo%
        (class object% (super-new)))
eval> (define Offener-Vorgang%
        (Bearbeitung-mixin
          (Akte-mixin Foo%)))
eval> (define OV001
        (new Offener-Vorgang%
          (id 9) (name "Erika Musterfrau")))
       ==> ERROR ...
       ;unused initialization arguments:
       ; (id 9) (name "Erika Musterfrau")
       ; for instantiated class
eval> (define OV001 (new Offener-Vorgang%))
eval> (send OV001 set-Bisheriger-Bearbeiter!
                  "Karl Meyer")
eval> (send OV001 get-Bisheriger-Bearbeiter)
      ==> "Karl Meyer"
eval> (get-field Aktuelle-Bearbeiter OV001)
      ==> "Fischer"
eval> (send OV001 get-Identifizierung)
      ==> ERROR ...
     ;send: no such method:
     ; get-Identifizierung for class
```

Ohne in der alltäglichen Programmierpraxis die Folgen ganz konkret durchdacht zu haben, wählt man häufig die Reihenfolge beim Mixen, die hier auch hätte sein können:

```
eval> (define Offener-Vorgang%
        (Akte-mixin
          (Bearbeitung-mixin Verwahrkonto%)))
eval> (define OV001
        (new Offener-Vorgang%
          (id 9) (name "Erika Musterfrau")))
eval> (send OV001 get-Identifizierung)
      ==> "Id: 9"
```

Wir nutzen nun beim Mixen die Überwachung der Einhaltung der Interfaces. Aus Gründen der Vereinfachung und damit dem leichteren Verstehen, setzen wir hier nur voraus, dass die spätere *Superclass* das Interface `Verwahrkonto-interface` (↪ S. 340) implementiert.

```
eval> (define Akte-mixin
        (mixin (Verwahrkonto-interface) ()
          (super-new)
                (define Aktenzeichen "XYZ")
                (define Standort "Archiv")
                (define/public get-Aktenzeichen
                  (lambda () Aktenzeichen))
                (define/public get-Standort
                 (lambda () Standort))
                (define/public set-Aktenzeichen!
                  (lambda (Text)
                   (set! Aktenzeichen Text)))
                (define/public set-Standort!
                  (lambda (Ort)
                   (set! Standort Ort)))))
eval> Akte-mixin
      ==> #<procedure:Akte-mixin>
```

Damit definieren wir analog wie schon vorher unsere Klasse `Offener-Vorgang%`. (↪ S. 342). Mit einer simplen Klasse `Foo%` zeigen wir, die Nichteinhaltung das Interfaces.

```
eval> (define Offener-Vorgang%
        (Bearbeitung-mixin
          (Akte-mixin Verwahrkonto%)))
eval> (define OV001
        (new Offener-Vorgang%
          (id 9) (name "Erika Musterfrau")))
eval> (send OV001 get-Identifizierung)
      ==> "Id: 9"
eval> (define Foo%
        (class object% (super-new)))
eval> (define Offener-Vorgang%
        (Bearbeitung-mixin
          (Akte-mixin Foo%)))
      ==> ERROR ....
     ;Akte-mixin: argument does not implement
     ; #<interface:Verwahrkonto-interface>:
     ; #<class:Foo%>
```

Traits

Ein *Trait* (≈ Eigenschaft, Wesenszug) ist ähnlich wie ein *Mixin*. Es kapselt auch eine Menge von Methoden, die dann einer Klasse hinzugefügt

werden. Ein *Trait* erlaubt jedoch, dass die einzelnen Methoden mit Operatoren (`trait`-Konstrukten) manipuliert werden können; z. B. mit dem `trait-sum`-Konstrukt, das die Methoden von zwei *Traits* mischt oder mit dem `trait-exclude`-Konstrukt, das eine Methode von einem *Trait* löscht oder mit dem `trait-alias`-Konstrukt, das eine Kopie einer Methode mit neuem Namen hinzufügt.

Der praktische Nutzen der *Traits* ist die Kombinationsmöglichkeit, selbst wenn gemeinsame Methoden betroffen sind. Dabei muss der Programmierer auftretende Kollisionen selbst, insbesondere durch *aliasing methods*, behandeln.

Hinweis: `trait`-Konstrukte.
Um in *PLT-Scheme* [21] mit der Entwicklungsumgebung von *DrScheme* (Version 4.2.3; Dez. 2009) die `trait`-Konstrukte nutzen zu können, bedarf es der `scheme/trait`-Bibliothek; d. h.:

```
eval> (require scheme/trait)
```

Zum Verdeutlichen der mannigfaltigen Möglichkeiten, die die `trait`-Konstrukte bieten, definieren wir zwei *Traits*: `Kontrolle-trait` und `Gebrauch-trait`. Dabei geht es um die Nutzung der Methode `get--Gebrauch`, die in beiden *Traits* deklariert ist.

```
eval> (define Kontrolle-trait
        (trait
         (define/public get-Gebrauch
           (lambda ()
             "Öffentlich"))
         (define/public kontrolliert
           (lambda ()
             "noch nicht"))))
eval> (define Gebrauch-trait
        (trait
         (field (Gebrauch "vertraulich"))
         (define/public get-Gebrauch
           (lambda ()
             Gebrauch))))
eval> (define Kontrolle+Gebrauch-trait
        (trait-sum
         (trait-exclude
          (trait-alias Kontrolle-trait
                       get-Gebrauch
                       get-Kontrolle-Gebrauch)
          get-Gebrauch)
         (trait-exclude
          (trait-alias Gebrauch-trait
                       get-Gebrauch
                       get-Gebrauch-Gebrauch)
```

[21] *PLT-Scheme* ↪`http://www.plt-scheme.org/` (Zugriff: 24-Oct-2009)

```
          get-Gebrauch)
         (trait
          (inherit get-Kontrolle-Gebrauch
               get-Gebrauch-Gebrauch)
          (define/public get-Gebrauch
            (lambda ()
              (string-append
               (get-Kontrolle-Gebrauch)
               (get-Gebrauch-Gebrauch)))))))
```

Mit dem Konstrukt `trait->mixin` konvertieren wir den Summen-*Trait* in ein `mixin`-Konstrukt, das wir dann wie bisher anwenden; also mit der Klasse `Offener-Vorgang%` (↪ S. 342) mixen.

```
eval> (define Vorgang-mixin
        (trait->mixin
          Kontrolle+Gebrauch-trait))
eval> (define Vorgang%
        (Vorgang-mixin Offener-Vorgang%))
eval> (define VG001
        (new Vorgang%
             (id 13)
             (name "Monika Meyer")))
eval> (send VG001 get-Gebrauch)
      ==> "Öffentlichvertraulich"
```

2.8.3 Zusammenfassung: Spezialisierung und Faktorisierung

Eine Klasse, ursprünglich auch Flavor genannt, definiert die Struktur und Eigenschaften einer Menge von Objekten, den Instanzen. Eine Operation, deklariert in der Klasse, heißt Methode. Sie wird über eine Nachricht an das Objekt aktiviert.

Bei der Vererbung kann eine Instanz die Struktur und Eigenschaften (Slots und Methoden) mehrerer Klassen übernehmen („erben“). Bei namensgleichen Slots und Methoden bedarf es einer Prioritätsregelung. Bei der dynamischen Vererbung werden Veränderungen des Vererbungsgrafen permanent überwacht und sofort berücksichtigt („propagiert“). Eine statische Vererbung wertet den Vererbungsgrafen nur zum Konstruktionszeitpunkt einer Instanz aus. Eine Modifikation des Grafen erfordert dann eine Neukonstruktion der Objekte.

Ein Aspekt zur Beurteilung einer objekt-orientierten Software-Entwicklungsumgebung sind die Optionen zur Spezialisierung und Faktorisierung. Sind „spezielle“ Eigenschaften, im Sinne von Teileinheiten, wie *Mixins* und *Traits*, jeder Zeit definierbar? Das Komplement zu dieser Spezialisierung ist die Faktorisierung. Hier geht es um die Möglichkeit, jederzeit Eigenschaften aus existierenden Subklassen zu „abstrakteren“ Einheiten zusammenfassen zu können.

Charakteristisches Beispiel für Abschnitt 2.8

Materialverwaltung Zum Verstehen der Aufgabe sind vorab einige Anforderungen und Testfälle formuliert. Dabei verweist die Abkürzung `A` auf eine *Anforderung* und die Abkürzung `T` auf einen *Testfall*.

`A1`: Das `Verwalten-mixin`-Konstrukt stellt die Methoden

- `A1.1`: `einlagern!` und
- `A1.2`: `entnehmen!` bereit.

`A2`: Die Klasse `Lagergut%` vererbt ihre Eigenschaften an Materialien, die aus den Klassen

- `A2.1`: `Stueckgut%`,
- `A2.2`: `Fluessigkeit%`,
- `A2.3`: `Eisenware%` und
- `A2.4`: `Farbe%`

spezialisiert sind.

`A3`: Die folgenden Klassen runden die Materialverwaltung ab.

- `A3.1`: `Lagerplatz%`,
- `A3.2`: `Bestellung%`,
- `A3.3`: `Lieferant%` und
- `A3.1`: `Magazin-Verwalter%`

`T1`: An die Instanz `E001` der Klasse `Eisenware%` werden folgende Nachrichten gesendet:

- `T1.1`: `eval> (send E001 einlagern!)`
- `T1.2`: `eval(send E001 entnehmen!)`

`T2`: Der aktuelle Bestand kann abgefragt werden, z. B. wie folgt:
`eval> (send` < *objekt* > `get-Bestand)`

```
eval> (define Lagergut-interface
        (interface ()
          get-Produkt-Identifikation
          get-Produkt-Bezeichnung
          get-Datum-letzte-Einlagerung
          get-Datum-letzte-Entnahme
          get-Bestand
          set-Produkt-Identifikation!
          set-Produkt-Bezeichnung!
          set-Datum-letzte-Einlagerung!
```

```
          set-Datum-letzte-Entnahme!
          set-Bestand!))
eval> (define Lagergut%
        (class* object% (Lagergut-interface)
          (init id bezeichnung)
          (define Produkt-Identifikation id)
          (define Produkt-Bezeichnung bezeichnung)
          (define Datum-letzte-Einlagerung 20100101)
          (define Datum-letzte-Entnahme 20100101)
          (define Bestand 0)
          (define/public get-Produkt-Identifikation
            (lambda () Produkt-Identifikation))
          (define/public get-Produkt-Bezeichnung
            (lambda () Produkt-Bezeichnung))
          (define/public get-Datum-letzte-Einlagerung
            (lambda () Datum-letzte-Einlagerung))
          (define/public get-Datum-letzte-Entnahme
            (lambda () Datum-letzte-Entnahme))
          (define/public get-Bestand
            (lambda () Bestand))
          (define/public set-Produkt-Identifikation!
           (lambda (Wert)
             (set! Produkt-Identifikation Wert)))
          (define/public set-Produkt-Bezeichnung!
            (lambda (Wert)
              (set! Produkt-Bezeichnung Wert)))
          (define/public set-Datum-letzte-Einlagerung!
            (lambda (Datum)
              (set! Datum-letzte-Einlagerung Datum)))
          (define/public set-Datum-letzte-Entnahme!
            (lambda (Datum)
              (set! Datum-letzte-Entnahme Datum)))
          (define/public set-Bestand!
            (lambda (Wert)
              (set! Bestand Wert)))
          (super-new)))

eval> (define Lieferant-mixin
        (mixin () ()
          (super-new)
          (field (Firmen-Name "unbekannt!"))
          (field (Lieferschein-Nummer 0))))

eval> (define Bestellung-mixin
        (mixin () ()
          (super-new)
          (field (Bestell-Vorgangs-Nummer 0))
          (field (Bestell-Datum 20100101))))

eval> (define Lagerplatz-mixin
        (mixin () ()
          (super-new)
          (field (Raum "Rm1"))
          (field (Regal "Rl1"))
          (field (Fach "Fh1"))
          (field
```

```
           (Kurzzeichen-Magazin-Verwalter
             "Vz"))))

eval> (define Stueckgut-mixin
        (mixin () ()
          (super-new)
          (field (Lager-Dimension "klein"))
          (field (Bestandswert 0))
          (define Kosten/Stueck 0)
          (define/public get-Kosten/Stueck
            (lambda ()
              Kosten/Stueck))
          (define/public set-Kosten/Stueck!
            (lambda (Wert)
              (set! Kosten/Stueck Wert)))))

eval> (define Eisenware-mixin
        (mixin () ()
          (super-new)
          (field (Rost-Freiheit #f))))

eval> (define Farbe-mixin
        (mixin () ()
          (super-new)
          (field (Verfalls-Datum 20100101))))

eval> (define Fluessigkeit-mixin
        (mixin () ()
          (super-new)
          (field (Behaelter-Klasse "I"))
          (field (Zuendtemperatur 0))))

eval> (define Magazin-Verwalter-mixin
        (mixin () ()
          (super-new)
          (define Name "")
          (field (Befugnis-Klasse "Alles"))
          (field (Kurzzeichen ""))
          (define/public get-Name
            (lambda ()
              Name))
          (define/public set-Name!
            (lambda(Text)
              (set! Name Text)))))

eval> (define Verwalten-mixin
      (mixin () ()
      (super-new)
      (define/public einlagern!
        (lambda ()
          (let ((Zugang (begin
                          (display
                            "Zugangsmenge: ")
                          (read)))
                (Zeitpunkt (begin
                             (display
```

```
                              "Datum JJJJMMTT: ")
                            (read)))
              (Kosten (begin
                        (display
                          "Kosten pro Stueck: ")
                        (read))))
          (send this set-Bestand!
                (+ (send this get-Bestand)
                   Zugang))
          (send this
                set-Datum-letzte-Einlagerung!
                Zeitpunkt)
          (send this
                set-Kosten/Stueck!
                (/ (+ (* (send this
                           get-Kosten/Stueck)
                         (send this
                           get-Bestand))
                      (* Kosten Zugang))
                   (+ (send this get-Bestand)
                      Zugang))))))
    (define/public entnehmen!
      (lambda ()
        (let ((Abgang (begin
                        (display
                         "Entnahmemenge: ")
                        (read)))
              (Zeitpunkt (begin
                           (display
                             "Datum JJJJMMTT: ")
                            (read))))
          (send this set-Bestand!
                (- (send this get-Bestand)
                   Abgang))
          (send this
                set-Datum-letzte-Entnahme!
                Zeitpunkt))))))
```

Im Folgenden werden Klassen zusammen „gemixt“ und exemplarisch Objekte erzeugt und Nachrichten versendet.

```
eval> (define Eisenware%
        (Eisenware-mixin
          (Verwalten-mixin
            (Stueckgut-mixin Lagergut%))))
eval> (define E001
        (new Eisenware%
               (id 'E001)
               (bezeichnung "Stahlmatte")))

eval> (send E001 einlagern!) ==>
Zugangsmenge: 50
Datum JJJJMMTT: 20100130
```

```
Kosten pro Stueck: 2.5
eval> (send E001 einlagern!) ==>
Zugangsmenge: 6
Datum JJJJMMTT: 20100131
Kosten pro Stueck: 2.0
eval> (send E001 entnehmen!) ==>
Entnahmemenge: 7
Datum JJJJMMTT: 20100131
eval> (send E001 get-Bestand) ==> 49
eval> (send E001 get-Kosten/Stueck)
      ==> 1.3225806451612903

eval> (define Farbe%
        (Farbe-mixin
          (Verwalten-mixin
            (Stueckgut-mixin
              (Fluessigkeit-mixin Lagergut%)))))
eval> (define F001
        (new Farbe%
             (id 'F001)
             (bezeichnung "rot")))
eval> (send F001 get-Bestand) ==> 0
eval> (send F001 set-Bestand! 100)
eval> (get-field Verfalls-Datum F001)
      ==> 20100101

eval> (define Magazin-Verwalter%
        (Magazin-Verwalter-mixin Lagergut%))
eval> (define MVW001
        (new Magazin-Verwalter%
            (id '?)
            (bezeichnung "?")))
eval> (send MVW001 set-Name! "Klaus Meyer")
eval> (send MVW001 get-Name)
      ==> "Klaus Meyer"
```

2.9 Kommunikation

Klar ist, oft bedingt die sogenannte „*Maschinisierung von Kopfarbeit*“ (↪ [136]) ein Zusammenwirken von mehreren bis zu quasi unzähligen Rechnern in einem Netzwerk. Eine der ersten Erfolgsgeschichten mit tausenden von Rechnern im Internet ist *SETI@home*.[22] Dabei handelt es sich um ein wissenschaftliches Experiment bei der Suche nach *Extraterrestrial Intelligence* (*SETI*, Mitwirkungsurkunde ↪ Abbildung 2.24 S. 352)

[22] *SETI@home* ↪ `http://setiathome.berkeley.edu/` (Zugriff: 3-Feb-2010)

Legende:
Mitwirkungsurkunde aus den Anfängen der Zusammenarbeit von sehr vielen Rechnern im Internet an einer gemeinsamen Aufgabe.

Abbildung 2.24: Kommunikation: Beispiel *SETI@home*

Für eine solche Kommunikation spielt zur Zeit das *Hypertext Transfer Protocol* (HTTP) eine dominierende Rolle. Klar ist auch, bei der hohen Leistungsfähigkeit der üblicherweise genutzten Rechner ist eine „Parallelverarbeitung“ oft sinnvoll. Stets sind Überlegungen angebracht, die Aufgabe so in Teile zu gliedern, dass diese auf mehrere Rechner im Netzwerk und/oder auf dem betroffenen Rechner „parallel“ abgearbeitet werden können. Beide Aspekte werden im Folgenden vertieft. Zunächst greifen wir mittels HTTP auf einen Web-Server zu (↪ Abschnitt 2.9.1 S. 352). Dann behandeln wir das *Thread*-System und zeigen dabei die Synchronisation von nebenläufigen Prozessen (↪ Abschnitt 2.9.2 S. 355).

2.9.1 HTTP-Kommunikation

Um auf ein Dokument im Web zugreifen zu können, muss sein *Uniform Resource Locator* (URL) in Form einer Zeichenkette (*String*) bekannt sein. Dieser String wird in eine URL-Struktur (`#<url>`) konvertiert und zwar mit dem Konstrukt `string->url`. Dann lässt sich das Dokument mittels `get-pure-port-` oder `get-impure-port`-Konstrukt öffnen. Das `get-pure-port`-Konstrukt liefert nur den Dokumenteninhalt. Das `get--impure-port`-Konstrukt zusätzlich auch den *Header* des Web-Dokumentes.

Um das *Hypertext Transfer Protocol* (HTTP) so nutzen zu können, ist die Bibliothek `net/url` erforderlich, d.h.:

```
eval> (require net/url)
```

Beispiel: Zugriff auf `http://www.hegb.de`

```
eval> (define MyInput-Port
       (get-pure-port
         (string->url "http://www.hegb.de")))

eval>(define Content
      (lambda ()
        (let ((zeile (read-line MyInput-Port)))
          (cond
           ((eof-object? zeile)
            (close-input-port MyInput-Port)
            zeile)
           (#t (display zeile)
               (newline)
               (Content))))))
eval> (Content) ==>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      lang="de" xml:lang="en">
<head>
...
</head>
<body>
...
</body>
</html>
#<eof>

eval> MyInput-Port ==>
      #<input-port:www.hegb.de>
eval> (read MyInput-Port) ==> ERROR ...
      ;read-char: input port is closed
```

Statt das Konstrukt `Content` zu definieren, hätte wir auch das eingebaute Konstrukt `display-pure-port` nutzen können. Allerdings verdeutlicht `Content` besser die *End-of-File*-Abfrage mittels `eof-object?` und die Notwendigkeit den *Port* wieder zu schließen.

Beispiel: `module Transfer`

Der folgende Modul `Transfer`[23] stellt das `Download`-Konstrukt bereit. Damit lassen sich auch `pdf`-Dateien aus dem Netz als Datei lokal speichern.

[23] Die entsprechende Datei `Transfer.ss` liegt im Verzeichnis:
`C:/Dokumente und Einstellungen/bonin/Anwendungsdaten/PLTScheme/4.2.3/collects/HEGB`

```
;;;;Datei aus dem Web auf die Platte laden
(module Transfer scheme
  (require net/url)
  (provide Download)
  (define Download
    (lambda (Web File)
      (letrec ((Web-Input-Port
                (get-pure-port (string->url Web)))
               (MyFile
                (open-output-file
                 File
                 #:exists
                 'replace))
               (Transfer
                (lambda ()
                  (let ((content
                         (read-byte
                          Web-Input-Port)))
                    (cond
                     ((eof-object? content)
                      (close-input-port
                       Web-Input-Port)
                      (close-output-port MyFile)
                      content)
                     (#t
                      (write-byte content MyFile)
                      (Transfer)))))))
        (Transfer)))))
```

Mit dem `Download`-Konstrukt, das auf der Basis von `read-byte` und `write-byte` arbeitet, transferieren wir z. B. den `pdf`-File[24] `waffeall.pdf` (≈ 1 MByte).[25] Die dazu insgesamt benötigte Transferzeit messen wir mit dem `time`-Konstrukt. Es gibt die Anzahl der Millisekunden an, die die CPU benötigt hat, die insgesamt zur Erzeugung des Resultates benötigt wurden sowie die darin enthaltene Zeit für *Garbage Collection* (gc).[26]

```
eval> (require HEGB/Transfer)
eval> (time
        (Download
          "http://www.hegb.de/waffe/waffeall.pdf"
          "D:\\bonin\\temp\\waffeall.pdf"))
```

[24] Das *Portable Document Format* ist ein plattformunabhängiges Dateiformat für Dokumente, das vom Unternehmen *Adobe Systems* entwickelt und 1993 veröffentlicht wurde. ↪ `http://www.adobe.com/devnet/pdf/pdf_reference.html` (Zugriff: 15-Jan-2010)

[25] Bei meiner Konfiguration (Notebook *lenovo Think Pad W500*) wurden für den Transfer von ≈ 1 MByte im Durchschnitt $< 1,5sec$ benötigt.

[26] Die Korrektheit der Zahlen hängt von der Plattform ab. Bei *Multithreading* kann die Zeitmessung Arbeit von anderen *Threads* enthalten.

```
==>
cpu time: 578 real time: 1328 gc time: 0
#<eof>
```

2.9.2 *Thread*-System

Ein *Thread*[27] (deutsch: Faden, Gedankengang) definiert einen „Ausführungsstrang“, d. h. die Reihenfolge der Ausführung bei der Abarbeitung eines Programms. *Threads* lassen sich unterscheiden in:

- *User Threads* ≡ Sie lassen Programm(teil)e miteinander verzahnt ablaufen. Ihre Funktionalität ist in einer User-Programmbibliothek abgebildet und nicht direkt im Betriebssystemkern implementiert.
- *Kernel Threads* ≡ *Threads*, die als Teil des Betriebssystems ablaufen. Ihre Funktionalität ist im Betriebssystemkern implementiert.

Da ein *User Thread*[28] dem Betriebssystem (quasi) unbekannt ist, muss der Programmierer mit Hilfe der bereitgestellten Konstrukte die Steuerung (*Scheduling*) selbst definieren. Während jedem *Prozess*[29] die Betriebsmittel und der Speicherraum zugeordnet sind, teilen sich *(User-)-Threads* innerhalb eines *Prozesses* die betriebssystemabhängigen Ressourcen (z. B. Dateien) und den Speicher. Der Koordinationsaufwand für *Threads* ist daher üblicherweise kleiner als für Prozesse. Beim *Thread*-Wechsel ist kein umfassender Kontextwechsel, wie bei einem Prozesswechsel notwendig, weil die *Threads* einen gemeinsamen Teil des Prozesskontextes verwenden.

Schritt für Schritt wird das `thread`-Konstrukt in Scheme erklärt. Dazu wird mit dem simplen Beispiel der Ausgabe der Buchstaben des Alphabetes begonnen (Idee für dieses didaktische Beispiel ↪ [186] S. 227–230).

Beispiel: Ausgabe des Alphabetes

Zur Ausgabe der kleinen und großen Buchstaben des Alphabetes definieren wir die beiden Konstrukte `Upcase-Alphabet` und `Downcase--Alphabet`.

```
eval> (define Upcase-Alphabet
        (lambda()
          (do ((i 65 (+ i 1)))
              ((> i 90) (newline))
```

[27] *Thread* ≡ „leichtgewichtiger“ Prozess oder Aktivitätsträger

[28] Auch als *User-level Thread* oder *Fiber* bezeichnet.

[29] Die Begriffswelt in der Informatik ist facettenreich. So wird der Begriff *Task* (deutsch: Aufgabe) auch als Synonym für *Prozess* verwendet. Dann spricht man vom *Multitasking* zur Abgrenzung zum *Multithreading*.

```
            (display (integer->char i)))))
eval> (Upcase-Alphabet) ==>
      ABCDEFGHIJKLMNOPQRSTUVWXYZ

eval> (define Downcase-Alphabet
        (lambda()
          (do ((i 97 (+ i 1)))
              ((> i 122) (newline))
            (display (integer->char i)))))
eval> (Downcase-Alphabet) ==>
      abcdefghijklmnopqrstuvwxyz
```

Die Prüfung der Iteration und die Typkonvertierung mit `integer->char` sind in beiden Konstrukten voneinander unabhängig. Diese Teile können daher nebenläufig durchgeführt werden. Nebenläufigkeit (engl.: *Concurrency*) bedeutet, ihre Abarbeitung kann in einem betrachteten Zeitraum voneinander unabhängig erfolgen. Für die Abarbeitung dieser Aufgaben von `Upcase-Alphabet` spielt es keine Rolle was gerade bei der Abarbeitung von `Downcase-Alphabet` passiert.

Zunächst definieren wir für `Upcase-Alphabet` und `Downcase-Alphabet` jeweils einen eigenen *Thread* (↪ S. 356). Das `thread`-Konstrukt appliziert ein *Thunk* (↪ S. 103) , d. h. ein `lambda`-Konstrukt ohne Parameter:

`eval> (thread <`*thunk*`>) ==> #<thread>`

mit:

<*thunk*>	≡ Enthält eingekleidet in ein `lambda`-Konstrukt ohne Parameter („nullstellig“) die abzuarbeitende Aufgabe.

Beispiel: `thread`

```
eval> (thread? (thread (lambda() 'Work)))
      ==> #t
eval> (thread? (thread (lambda() (display 'Work))))
      ==> #t
          Work
```

Deutlich wird an diesem Beispiel, dass der Wert des `thread`-Konstruktes nicht der „*Body*“-Wert des *Thunk* ist. Es bedarf daher eines Nebeneffektes (hier mit `display`), wenn der *Thread* eine Wirkung nach außen zeigen soll.

```
eval> (define Alphabet-with-Threads
        (lambda ()
          (thread
            (lambda () (Upcase-Alphabet)))
          (thread
            (lambda () (Downcase-Alphabet)))
          'end))
```

```
eval> (Alphabet-with-Threads) ==>
      end
      abcdABCefghijklmnopqrstuvwxyz
      DEFGHIJKLMNOPQRSTUVWXYZ

eval> (Alphabet-with-Threads) ==>
      end
      abcdAefghijklmnopqrBCDEFGHstuvwxyz
      IJKLMNOPQRSTUVWXYZ
```

Erwartungsgemäß ist das Resultat beim mehrmaligen Evaluieren des Konstruktes `Alphabet-with-Threads` unterschiedlich. Bei der Ausgabe mittels `display` kommen die beiden nebenläufig arbeitenden *Threads* je nach den aktuellen Gegebenheiten jeweils unterschiedlich schnell vorwärts. Um diesen „Buchstabensalat“ zu vermeiden und die Ausgabe von `end` auch an des Ende zu setzen, wird ein Thread mit dem `sleep`-Konstrukt zum Warten veranlasst.

```
eval> (void? (sleep < secs >))
              ==> #t
```

mit:

$<$ *secs* $>$	$\equiv$ *Thread* wartet die nicht-negative Angabe von Sekunden.

Mit dem `break-thread`-Konstrukt unterbrechen wir die Abarbeitung des jeweiligen *Threads*. Das `current-thread`-Konstrukt hat den aktuellen *Thread* als Wert.

```
eval> (current-thread)
              ==> #<thread:handler>
```

```
eval> (define Alphabet-with-Threads
        (lambda ()
          (thread
            (lambda () (Upcase-Alphabet)))
          (thread
            (lambda () (Downcase-Alphabet)))
          (sleep .001)
          (break-thread (current-thread))
          'end))
eval> (Alphabet-with-Threads) ==>
      abcdefghijklmn.op A. user break
      BCDqrstuvwxyz
      EFGHIJKLMNOPQRSTUVWXYZ

eval> (Alphabet-with-Threads) ==>
```

```
      abcdefghijklmnopqrstuvwxyz
      ABCDEFGH. IJKLMNOPQRSTUVWXYZ
      . user break
```

Das obige Ergebnis zeigt, dass die Unterbrechung für den (Haupt-)-*Thread* die Ausgabe von `end` verhindert. Zum Nachweis, dass die beiden Buchstabenausgabe-*Threads* gleichbehandelt werden, definieren wir wie folgt:

Konstrukt `Alphabet-with-Threads`

```
eval> (define Alphabet-with-Threads
       (lambda ()
         (letrec (
             (Upcase-Alphabet
              (lambda ()
                (do ((i 65 (+ i 1)))
                    ((> i 90) (newline))
                  (display (integer->char i))
                  (sleep 0.1))))
             (Downcase-Alphabet
              (lambda ()
                (do ((i 97 (+ i 1)))
                    ((> i 122) (newline))
                  (display (integer->char i))
                  (sleep 0.1))))
             (th1
              (thread
               (lambda () (Upcase-Alphabet))))
             (th2
              (thread
               (lambda () (Downcase-Alphabet)))))
       (sleep 3)
       'end)))
eval> (Alphabet-with-Threads) ==>
aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrRsStTuUvVwWxXyYzZ

end
```

Synchronisation

Da beim *User-Thread* die Steuerung vom Programmierer definiert werden muss, ist beispielsweise der Zugriff auf eine gemeinsame Ressource genau zu planen. Für das Verstehen einer solchen Synchronisation verwenden wir als Beispiel den Namen eines Kontos, der durch die zweimalige Applikation des Mutators `set!` entsteht. Zunächst sorgen wir ohne *Threads* für die sequentielle Abarbeitung wie folgt:

```
eval> (define Konto-Name
        (lambda ()
          (let* ((name 'Ausgaben))
            (set! name
              (string->symbol
               (string-append
                (symbol->string name)
                "I")))
            (set! name
              (string->symbol
               (string-append
                (symbol->string name)
                "dubios")))
        (display name))))

eval> (Konto-Name) ==>
AusgabenIdubios
```

Die beiden `set!`-Konstrukte werden in *Thunks* „verpackt“ und jeweils einem `thtread`-Konstrukt zugeordnet. Wie im Beispiel `Alphabet-with-Threads` (↪ S. 358) steuern wir mit `sleep`-Konstrukten indirekt die „Prioritäten“ der *Threads*.

```
eval> (define Konto-Name-with-Threads
        (lambda (x y z)
          (let ((name 'Ausgaben))
            (thread
             (lambda ()
               (set! name
                 (string->symbol
                  (string-append
                   (begin (sleep x)
                          (symbol->string name))
                          "I")))))
            (thread
             (lambda ()
               (set! name
                 (string->symbol
                  (string-append
                   (begin (sleep y)
                          (symbol->string name))
                   "dubios")))))
        (sleep z)
        (display name))))
```

Mit unterschiedlichen Sekundenangaben für die Dauer der Wartezeit erhalten wir folgende Ergebnisse. Deutlich wird dabei, dass mit hohen `sleep`-Werten eigentlich wieder eine sequentielle Abarbeitung erzwungen wird.

```
eval> (Konto-Name-with-Threads 0.0 0.0 0.5)
       ==> AusgabendubiosI
eval> (Konto-Name-with-Threads 0.3 0.0 0.5)
      ==> AusgabendubiosI
eval> (Konto-Name-with-Threads 0.0 0.3 0.5)
      ==> AusgabenIdubios
```

Mutal Exclusion

Bei der Nebenläufigkeit der *Threads* wollen wir den simultanen Zugriff auf eine gemeinsame Ressource, wie hier die lokale Variable `name`, vermeiden. Der kritische Abschnitt ist jeweils die Abarbeitung des Mutators (`set!`). Dazu nutzen wir einen einfachen Mechanismus zur Synchronisation, genannt *Mutex Exclusion*[30] (deutsch: gegenseitiger Ausschluss).

Beispiel: Mehrere Ein- und Auszahlungen auf ein Konto

Wir nehmen nun an, auf einem Konto, hier `A001`, der Klasse `Konto%` sind Ein- und Auszahlungen zu buchen. Wenn die Deckung fehlt, ist eine Auszahlung abzuweisen (↪ dazu das Kontobeispiel in Abschnitt 2.8.1 S. 329). Zunächst versuchen wir wieder mit dem `sleep`-Konstrukt zu steuern; hier `Konto-Name-with-Threads` (↪ S. 359). Damit eine Einzahlung vor einer Auszahlung erfolgt lassen wir die Auszahlung (Methode `abheben!`) länger warten.

Klasse `Konto%`

```
eval> (define Konto%
        (class object%
          (define Kontostand 0)
          (super-new)
          (define/public get-Kontostand
            (lambda () Kontostand))
          (define/public abheben!
            (lambda(Betrag)
              (sleep 0.3)
              (cond ((<= Betrag Kontostand)
                 (set! Kontostand
                        (- Kontostand Betrag))
```

[30] *Yoah Bar-David* bemerkt dazu: *„The mutual exclusion problem was formally defined by Edsger Dijkstra in 1965. Since then many solutions were published, one of them is a simple solution by Gary Peterson in 1981. Information on the problem, known solutions, and related problems can be found in many books and web sites about distributed algorithms or operating systems.“*
(↪ `http://bardavid.com/mead/` (Zugriff: 23-Jan-2010))
Eine Petri-Netz-Darstellung zur *Mutex Exclusion* findet man beispielsweise unter: ↪ `http://www.cs.adelaide.edu.au/users/esser/mutual.html` (Zugriff: 23-Jan-2010)

```
                (printf
                  "Neuer Kontostand: ~s EUR.~n"
                  Kontostand))
                (#t (printf
                      "Keine Deckung! ~n")))))
          (define/public einzahlen!
            (lambda (Betrag)
              (sleep 0.1)
              (set! Kontostand
                      (+ Kontostand Betrag))
              (printf
                "Neuer Kontostand: ~s EUR.~n"
                Kontostand)))))
```

Klar ist, dass ohne *Threads* bei den Kontobewegungen die Reihenfolge zwischen `abbuchen!` und `einzahlen!` nicht beeinflusst wird. Beim folgenden Fall wird daher auf keine Deckung verwiesen, obwohl gleich danach eine ausreichende Einzahlung erfolgt.

```
eval> (define A001 (new Konto%))

eval>(define Batch-Bewegungen
        (lambda ()
          (send A001 abheben! 100)
          (send A001 einzahlen! 200)))
eval> (Batch-Bewegungen) ==>
      Keine Deckung!
      Neuer Kontostand: 200 EUR.
```

Den gleichen Fall verarbeiten wir nun mit *Treads* und erhalten für diese Daten ein korrektes Ergebnis:

```
eval> (define Batch-Bewegungen-with-Threads
        (lambda ()
          (thread
            (lambda ()
              (send A001 abheben! 100)))
          (thread
            (lambda ()
              (send A001 einzahlen! 200)))
      (sleep 1.0)))

eval> (define A001 (new Konto%))

eval> (Batch-Bewegungen-with-Threads) ==>
Neuer Kontostand: 200 EUR.
Neuer Kontostand: 100 EUR.
```

Die Synchronisation ist aber für mehre Buchungsvorgänge nicht passend. Nehmen wir jetzt die folgenden 5 Buchungen an, dann ist das Ergebnis falsch.

```
eval> (define Batch-Bewegungen-with-Threads
        (lambda ()
          (thread
            (lambda ()
              (send A001 abheben! 100)))
          (thread
            (lambda ()
              (send A001 abheben! 100)))
          (thread
            (lambda ()
              (send A001 einzahlen! 200)))
          (thread
            (lambda ()
              (send A001 abheben! 100)))
          (thread
            (lambda ()
              (send A001 einzahlen! 200)))
          (sleep 1.0)))

eval> (define A001 (new Konto%))
eval> (Batch-Bewegungen-with-Threads) ==>
      Neuer Kontostand: 200Neuer Kontostand: 400 EUR.
      EUR.
      Neuer Kontostand: 300 EUR.
      Neuer Kontostand: 200Neuer Kontostand: 100 EUR.
      EUR.
```

Das Ergebnis zeigt, das zu gewährleisten ist, das ein einmal begonnener Buchungsvorgang solange nicht unterbrochen wird, bis seine Ausführung (im kritischen Pfad) vollständig abgeschlossen ist. Die Ausführungsdauer — und damit die definierte Wartedauer der anderen Zugreifer — darf dabei keine Rolle spielen. Benötigt wird ein *Monitor*, der den Zugriff durch die Verwaltung einer Sperre (bzw. einer Freigabe) steuert, d. h. eine „serialisierte Methode“ ist explizit zu sperren bzw. freizugeben. Dazu wird ein zusätzliches Objekt, das analog zu einer Rot-Grün-Ampel arbeitet, erforderlich.

Der benötigte Serialisierer kann mit *Semaphoren* (deutsch: Signalmast) definiert werden. Ein *Semaphore* hat einen internen Zähler. Wenn der Zählerwert gleich Null ist, so blockiert der *Semaphore* die Ausführung des *Threads* solange, bis ein anderer *Thread* den Zähler inkrementiert, d. h. das `semaphore-post`-Konstrukt evaluiert.

Ein solcher Serialisierer synchronisiert die *Threads* einer Gruppe. Da die Anzahl der Gruppenmitglieder nicht bekannt ist, nutzen wir hier die Möglichkeit an eine `lambda`-Variable beliebig viele Werte zu binden, wie folgt:

```
eval> (define a (lambda args args))
eval> (a 'd 'e) ==> (d e)
```

Auf der Basis von `make-semaphore`, `semaphore-wait` und `semaphore-post` definieren wir nun die notwendige „Steuerungsampel“.

```
eval> (define Synchronisation
        (lambda ()
          (let ((Control-Signal (make-semaphore 1)))
            (lambda (fun)
              (let ((Serialized-Function
                  (lambda args
                    (semaphore-wait Control-Signal)
                    (let ((Value (apply fun args)))
                      (semaphore-post Control-Signal)
                      Value))))
                Serialized-Function)))))
```

Mit dieser *Mutual Exclusion* wird nun das gewünschte Ergebnis erzielt:

```
eval> (define Batch-Bewegungen-with-Threads
    (lambda ()
      (let ((Sync (Synchronisation)))
      (thread (Sync
        (lambda () (send A001 abheben! 100))))
      (thread (Sync
        (lambda () (send A001 abheben! 100))))
      (thread
        (Sync (lambda () (send A001 einzahlen! 200))))
      (thread
        (Sync (lambda () (send A001 abheben! 100))))
      (thread
        (Sync (lambda () (send A001 einzahlen! 200))))
      (sleep 1.0))))
eval> > (define A001 (new Konto%))
eval> (Batch-Bewegungen-with-Threads)
      Neuer Kontostand: 200 EUR.
      Neuer Kontostand: 100 EUR.
      Neuer Kontostand: 0 EUR.
      Neuer Kontostand: 200 EUR.
      Neuer Kontostand: 100 EUR.
```

Bei genauer Betrachtung wird deutlich, dass das klassische *Producer/-Consumer*-Problem mit diesem simplen Konstrukt `Synchronisation` nicht für alle Datenfälle gelöst ist. Es bedarf dazu einer Kommunikation zwischen den *Threads*, so dass ein *Thread* für `abbuchen!` bei keiner Deckung „nachfragt“, ob es noch *Threads* für `einzahlen!` gibt (↪ S. 364). Bei einer solch aufwendigen Steuerung, stellt sich die Frage, ob eine vorhergehende Sortierung der Batch-Daten nicht zweckmäßiger ist.

Mit dem `thread-wait`-Konstrukt lassen wir abhängig von einer gegebenen Situation den anderen *Thread* zunächst fertig abarbeiten.

```
eval> (void? (thread-wait <thread>))  ==> #t
```

mit:

< thread >	≡ Blockiert den aktuellen *Thread* solange bis *< thread >* terminiert hat. Hinweis: `(thread-wait (current-thread))` ist ein *Deadlock*

Damit wird nun eine einfache Kommunikation zwischen den Threads für das Einzahlen und Abheben definiert. Abhängig vom Prüfergebnis, ob Deckung nicht gegeben ist, wartet der *Thread* fürs Abheben (`t-abheben`) auf die Terminierung des *Threads* fürs Einzahlen (`t-einzahlen`).

```
eval> (define Bewegungen-with-Thread-Communication
  (lambda (einzahlen abheben)
    (let* ((t-einzahlen
            (thread
             (lambda ()
               (send A001 einzahlen! einzahlen))))
           (t-abheben
            (thread
             (lambda ()
               (if (<=
                    (send A001 get-Kontostand)
                    abheben)
                 (thread-wait t-einzahlen)
                 'Ok)
               (send A001 abheben! abheben)))))
      (sleep 1.0))))

eval> (define A001 (new Konto%))
eval> (Bewegungen-with-Thread-Communication 200 100)
      Neuer Kontostand: 200 EUR.
      Neuer Kontostand: 100 EUR.
eval> (Bewegungen-with-Thread-Communication 50 150)
      Neuer Kontostand: 150 EUR.
      Neuer Kontostand: 0 EUR.
eval> (Bewegungen-with-Thread-Communication 0 100)
      Neuer Kontostand: 0 EUR.
      Keine Deckung!
```

Channel

In *PLT-Scheme* bietet das eingebaute Konzept *Channel* (Kanal) eine Möglichkeit die *Threads* zu synchronisieren und sie untereinander Werte austauschen zu lassen. Es gibt zwei *Channel*-Kategorien:

1. *Synchronous Channels*

2. *Buffered Asynchronous Channels*

Im ersten Fall (*synchronous channels*) wird mit dem `channel-get`-Konstrukt solange blockiert bis der Sender in der Lage ist, den Wert über den Kanal bereitzustellen. Mit dem `channel-put`-Konstrukt wird blockiert bis der Empfänger in der Lage ist, den Wert zu akzeptieren. Im zweiten Fall (*buffered asynchronous channels*) wird eine zeitliche Entkopplung bewirkt. Mit dem folgenden Beispiel `Server` wird die Arbeitsweise der *asynchronous channels* skizziert.[31] Dazu bedarf es der *PLT-Scheme*-Bibliothek `async-channel`, d. h. :

```
eval> (require scheme/async-channel)
```

Den *Thread* konstruieren wir in einem Namensraum mit den beiden (lokalen) `lambda`-Variablen `Input-Channel` und `Output-Channel` wie folgt:

```
eval> (define Server
        (lambda (Input-channel Output-channel)
          (thread
            (lambda ()
              (define get
                (lambda ()
                  (async-channel-get Input-channel)))
              (define put
                (lambda (x)
                  (async-channel-put Output-channel x)))
              (define do-long-computing
                (lambda () ;Some work
                  (* (sqrt 7.0)(sqrt 7.0))))
              (let loop ((data (get)))
                (case data
                  ((add) (begin
                           (put (+ 1 (get)))
                           (loop (get))))
                  ((long) (begin
                            (put
                              (do-long-computing))
                            (loop (get))))
                  ((quit) (void))))))))
```

Hinweis: `let`-Konstrukt mit „Sprungmarke“.
Hier nutzen wir das `let`-Konstrukt in folgender Ausprägung:

`eval> (let` $<$ *proc* $-$ *id* $>$ `({(`$<$ *id* $><$ *init* $-$ *expr* $>$`})`
$<$ *body* $>$`)`

mit:

$<$ *proc* $-$ *id* $>$ $\equiv$ Innerhalb von $<$ *body* $>$ an das `let`-Konstrukt selbst gebunden.

[31] Idee der *PLT-Scheme*-Dokumentation entnommen:
↪ `file:///C:/Programme/PLT/doc/reference/async-channel.html` (Zugriff: 2-Feb-2010)

Mit dem Konstruktur `make-async-channel` erzeugen wir zwei *asynchronous Channels* und binden diese an die `lambda`-Variablen von `Server`.

```
eval> (define ->Server (make-async-channel))
eval> (define <-Server (make-async-channel))
eval> (async-channel? ->Server) ==> #t

eval> (Server ->Server <-Server) ==> #<thread>
```

Mit dem `async-channel-put`-Konstrukt schicken wir zwei Werte, hier `'add` und 6, in den *Channel* `->Server`, um dann das Resultat 7 über den *Channel* `<-Server` zu erhalten.

```
eval> (async-channel-put ->Server 'add)
eval> (async-channel-put ->Server 6)
eval> (printf "Result: ~a\n"
        (async-channel-get <-Server))
      ==> Result: 7
```

Wir beschäftigen jetzt `->Server` mittels `async-channel-put` und können derweil andere Dinge tun, hier ein nur `(display 'Working!)`. Dann fragen wir das Ergebnis nach und beenden diese *Channel*-Verknüpfung.

```
eval> (async-channel-put ->Server 'long)
eval> (display 'Working)
      ==> Working
eval> (printf "Server-Computation: ~a\n"
        (async-channel-get <-Server))
      ==>
      Server-Computation: 7.000000000000001

eval> (async-channel-put ->Server 'quit)
```

Exkurs: *Parameter*

In *PLT-Scheme* sind *Parameter* ein abgeleitetes Konzept, das auf *Thread*-Zellen im Verbund mit *Continuation Marks* (↪ Abschnitt 2.5.2 S. 285) basiert. Mit dem Konstruktor `make-parameter`, dem Prädikat `parameter?` und weiteren Konstrukten können wir eigene Parameter definieren und nutzen. Erfreulicherweise fragen einige primitive Konstrukte, wie z.B. das `eval`-Konstrukt, bestimmte Parameter ab. Das folgende Beispiel skizziert einen solche Parametrisierung (Näheres ↪ *PLT-Scheme* Online-Dokumentation).

```
eval> make-parameter
      ==> #<procedure:make-parameter>
eval> (make-parameter 1)
      ==> #<procedure:parameter-procedure>
eval> (parameter? current-namespace)
      ==> #t
```

```
eval> (define Foo  'top-level)
eval> (let ((ns (make-base-namespace)))
        (parameterize ((current-namespace ns))
          ;Im Namespace ns wegen Parameter-
          ; abfrage von eval
          (eval '(define Foo 'let-level))
          (display (eval 'Foo))
          (newline)
          (display Foo)))
      ==>
      let-level
      top-level
```

Exkurs: *Custodian* (≈ Verwalter, Aufpasser)
In *PLT-Scheme* übernimmt das Verwalten einer Sammlung von *Threads*, *File-Stream Ports*, *TCP Ports*[32], *TCP Listeners*, *UDP Sockets*[33], und *byte converters* ein *Custodian*. Immer wenn z. B. ein *Thread* keriert wird, kommt er unter die Verwaltung des *current custodian*. Jeder *custodian* (Ausnahme *root custodian*) wird selbst durch einen *custodian* verwaltet, so das die *custodians* eine Hierarchie bilden.

Dem Programmierer stehen mit den entsprechenden Konstrukten vielfältige Steuerungsoptionen für *Threads* zur Verfügung, wie die folgenden Zeilen andeuten:

```
eval> (define Foo (make-custodian))
eval> Foo ==> #<custodian>
eval> (custodian? Foo) ==> #t
eval> (custodian-shutdown-all Foo)
eval> (void? (custodian-shutdown-all Foo))
      ==> #t
```

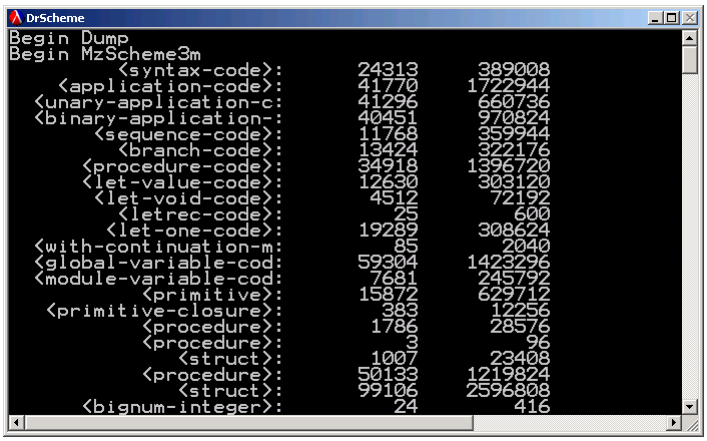
Mit den Mitteln *Parameters* und *Custodians* lassen sich *Threads* beeinflussen. Bei einer so „tiefen“ Steuerung durch den Programmierer wird häufig die Frage nach dem Speicherplatz und dessen Verwendung relevant. Dazu gibt es ebenfalls Konstrukte, z. B. `(current-memory-use)` oder `(dump-memory-stats)`. Ein Beispiel für die Statistik über den Speicher zeigt Abbildung 2.25 S. 368.

2.9.3 Zusammenfassung: Kommunikation

Die Kommunikation über das *Hypertext Transfer Protocol* (HTTP) ist leicht definierbar, denn dazu dient das *Port*-Sytem (*Port* ≡ *produce and consume bytes*). Die HTTP-Abbildung erfolgt daher analog zum Arbeiten mit Input- und Output-Files.

Für die Gestaltung von nebenläufig ablaufenden Programm(teil)en gibt es ein *Threads*-System. Die einzelnen *Threads* lassen sich synchronisieren. Ein einfacher Mechachnis dafür ist der gegenseitige Ausschluss

[32] *TCP* ≡ *Transmission Control Protocol* — das dominante Netzwerkprotokoll im Internet.
[33] *UDP* ≡ *User Datagram Protocol* — Transportprotokoll der Internet-Protokollfamilie.



Legende:
`eval> (dump-memory-stats)==> ;` Auszug diese Abbildung
`eval> (current-memory-use) ==> 130409780;` Bytes

Abbildung 2.25: Beispiel: Speicherauszug

(*Mutal Exclusion*), der mit den bereitgestellten *Semaphoren* einfach realisierbar ist. Zusätzlich gibt es synchrone und asynchrone *Channels* zum Austausch von Werten zwischen *Threads*.

Charakteristische Beispiele für Abschnitt 2.9

http-Kommunikation

```
eval> (define get-Web
        (lambda (Adresse)
          (letrec
            ((MyInput-Port
              (get-pure-port
               (string->url Adresse)))
             (Content
              (lambda ()
                (let ((zeile
                       (read-line MyInput-Port)))
                  (cond
                   ((eof-object? zeile)
                    (close-input-port
                     MyInput-Port)
                    zeile)
                   (#t
                    (display zeile)
                    (newline)
```

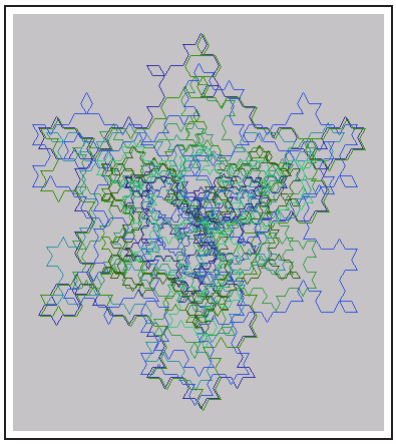
```
                    (Content)))))))
      (Content))))

eval> (get-Web "http://www.zeit.de") ==>
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML><HEAD>
<TITLE>301 Moved Permanently</TITLE>
</HEAD><BODY>
<H1>Moved Permanently</H1>
The document has moved
<A HREF="http://www.zeit.de/index">here</A>.<P>
<HR>
<ADDRESS>Apache/1.3.34 Server at www.zeit.de Port 80</ADDRESS>
</BODY></HTML>
#<eof>
```

Thread-Kommunikation

```
eval> (define Work
  (lambda ()
    (let* ((fun
            (lambda (x)
              (cond
               ((equal? x 'Bahn) (display "Ok! "))
               (#t (display x) (display "->")))))
           (task-Meyer
            (thread
             (lambda ()
               (sleep 0.5)
               (for-each
                fun
                '(Lager Hof Haus Bahn)))))
           (task-Schulze
            (thread
             (lambda ()
               (thread-wait task-Meyer)
               (for-each
                fun
                '(Garten Bahn Auto Lager))))))
      (sleep 1)
      'done)))
eval> (Work) ==>
      Lager->Hof->Haus->Ok! Garten->Ok! Auto->Lager->done
```

Kapitel 3

Konstruktionsempfehlungen



Das Primärziel der Softwarekonstruktion ist eindeutig und unstrittig. Es ist:

- *nützliche* Software zu konstruieren,
  d. h. Software, die eine nützliche Aufgabe erledigt, wobei die Erledigung vereinfacht oder erst ermöglicht wird,
- und zwar *hinreichend fehlerfrei* und *termingerecht*.

Von diesem Primärziel leiten sich eine Vielzahl von Sekundärzielen ab, wie z. B.:

1. Konstruiere Programm(teil)e in gewünschter Qualität.
2. Konstruiere Programm(teil)e, die wiederverwendbar sind.
3. Konstruiere Programm(teil)e, die sich pflegen und ergänzen lassen.

4. Konstruiere Programm(teil)e, die dokumentierbar und durchschaubar sind.

5. ...

Diese Sekundärziele sind kein Selbstzweck. Wir fordern daher z. B. keine qualitativ hervorragende Dokumentation, sondern eine „angemessene", also am ökonomischen Primärziel *Nützlichkeit* gemessene Dokumentation. Die Dokumentation ist stets, auch im juristischen Sinne, wesentlicher Bestandteil der Konstruktion. Software ohne Dokumentation ist vergleichbar mit einer Wohnung ohne Fenster. Fehlt ein wesentlicher Bestandteil, dann ist die Übergabe unvollständig und das „Produkt" kaum nutzbar. Ein wesentlicher Bestandteil ist (häufig) ein wesentlicher Kostenfaktor. Daher wollen wir aus Kostenerwägungen das einmal „Erarbeitete" möglichst mehrfach nutzen.

Aus diesem Wiederverwendungsziel hat sich das *Baukasten-Konstruktionsprinzip* entwickelt. Dieses in vielen Ingenieurdisziplinen angewendete Prinzip fordert, dass eine Konstruktion aus möglichst wenigen standardisierten Bausteinen zu realisieren ist.

Das Baukasten-Konstruktionsprinzip hat einen analytischen und einen synthetischen Aspekt. Das analytische Problem lautet: Wie kann eine gegebene Konstruktion durch eine Konstruktion gleicher Leistung ersetzt werden, die nur Konstrukte aus der vorgegebenen Konstruktemenge des genormten Baukastens verwendet? Das synthetische Problem lautet: Welche Konstrukte aus der vorgegebenen Menge werden benötigt, und welche Verknüpfungen dieser Konstrukte sind vorzunehmen, um eine Konstruktion mit der spezifizierten Leistung zu erhalten? Beide Aspekte betreffen die gesamte Konstruktion einschließlich Spezifikation und Dokumentation, also nicht nur die Konstruktion des Quellcodetextes.

In diesem Kapitel diskutieren wir Empfehlungen für eine durchschaubare (*transparente*) Dokumentation (↪ Abschnitt 3.1 S. 373). Fragen der Benennung von Konstrukten (↪ Abschnitt 3.1.1 S. 377) und der Kommentierung im Quellcodetext werden vertieft (↪ Abschnitt 3.1.2 S. 389). Dabei erörtern wir die Bezugnahme auf vorhergehende und nachfolgende Dokumente (↪ Abschnitt 3.1.2 S. 389). Ein kleines Regelsystem verdeutlicht diese Empfehlungen (↪ Abschnitt 3.1.4 S. 397). Es weist Grundzüge der Konstruktion von sogenannten Produktionssystemen auf.[1]

LISP ist — wie jede höhere Programmiersprache — zur Spezifikation einer Aufgabe einsetzbar. Wir können mit LISP nicht nur die Lösung einer Aufgabe formulieren, sondern auch den Lösungsraum beschreiben. Die Spezifikation des Lösungsraums befasst sich, etwas salopp formuliert, mit dem Problem: *Was gesollt werden soll?* (kurz: mit dem *Was*), d. h. mit der deskriptiven Aufgabenpräzisierung. Die Programmierspra-

[1] Z. B. im Sinne von *OPS5*, ↪ [65]

che im engeren Sinne zielt auf das Problem, wie das Spezifizierte realisiert wird (kurz: auf das *Wie*). Die Grenze zwischen *Was* und *Wie* ist fließend. Das realisierbare *Was* enthält in der Praxis (mindestens implizite) *Wie*-Aussagen.

Eine Programmiersprache kann die Grenze in Richtung Lösungsraumbeschreibung zumindest im Umfang einer Pseudocode-Notation verschieben. Der Vorteil einer Pseudocode-Notation liegt in ihrer Flexibilität und der Einsatzmöglichkeit über mehrere Projektphasen. Sämtliche Eigenschaften und Merkmale eines Programms sind jedoch so nicht spezifizierbar, z. B. können keine Aussagen zur Antwortzeit, zum Betriebsmittelbedarf oder zu Realtime-Eigenschaften notiert werden. Programmiersprachen ersetzen daher nicht spezielle Spezifikationssprachen[2].

Die Empfehlungen zum Spezifizieren mit LISP basieren auf einer Beschreibung der Import- und Export-Schnittstellen. Ihre schrittweise Verfeinerung und Präzisierung wird anhand einer Checkliste vollzogen (↪ Abschnitt 3.2.2 S. 411). Vorab erörtern wir die Randbedingungen für das Spezifizieren in LISP-Notation (↪ Abschnitt 3.2.1 S. 407).

Im objekt-orientierten Paradigma („Denkmodell“) ist die Notation in *Unified Modeling Language* (UML) üblich und zweckmäßig. Wir erörtern daher das UML-Klassendiagramm. Dabei nutzen wir Grafik-Optionen um UML-Symbole mit Bezug zu unseren Beispielen zu programmieren (↪ Abschnitt 3.3 S. 418). Mit den Kenntnissen von Grafik-Konstrukten erörtern wir ein Tool zum Präsentieren (↪ Abschnitt 3.4 S. 437). Dabei realisieren wir eine *Slideshows* zum Überzeugen von Programmierern, die bisher andere Programmiersprachen nutzten, von der „Nützlichkeit von LISP“ (↪ Abschnitt 3.4.1 S. 438).

3.1 Transparenz der Dokumentation

Konstruieren heißt dokumentieren. Das Ergebnis dieser Tätigkeit, die Konstruktion, ist ebenfalls ein Dokument. Softwarekonstruktion bedeutet, eine vorgegebene Kette von Dokumenten zu produzieren. Das Kostenbudget und die Terminvorgaben bedingen, dass wir diese dominierende Aufgabe nicht missinterpretieren: *Je mehr Dokumente produziert werden, umso besser.* Häufig gibt es zu viele (unnütze, nicht zweckorientierte) Dokumente (↪ z. B. [163], S.205).

Bei der Dokumentation geht es nicht um die Menge, sondern um die Qualität, d. h. um die Transparenz in Bezug auf ein schnelles und hinreichend genaues Verstehen der Konstruktion.

Die populäre Dokumentationsstrategie orientiert sich an der Maxime: „Erst Entwurf, dann Reinschrift!“ Damit ist Dokumentieren stets eine nachträgliche Arbeit. Die Dokumentation, die erst nach vollbrachter Tat,

[2] Z. B. *SLAN-4*, ↪ [10]

d. h. nach Abschluss der eigentlichen Konstruktion, entsteht, vergeudet Zeit und ist (fast) nutzlos. Zu solchen Nacharbeiten zählen insbesondere (↪ z. B. [153], S. 140):

1. die verbale Beschreibung der Kontrollstruktur,
2. Programmablaufpläne (*PAP*'s) und
3. das Einfügen von Kommentaren.

Ab einer gewissen Komplexität der Konstruktionsaufgabe sind:

1. vorbereitende Dokumente (Lastenheft, Pflichtenheft, Begründungen für das gewählte Konzept etc.),
2. Konstruktionsbegleitende Dokumente (Testprotokolle, Integrationspläne etc.) und
3. nachbereitende Dokumente (Wartungsrichtlinien, Installationsanweisungen etc.)

zu erstellen. Wir untergliedern daher die Dokumentation in Dokumente, die das Ergebnis (die Software im engeren Sinne) beschreiben und in Dokumente, die den Prozess ihrer Erstellung nachweisen.

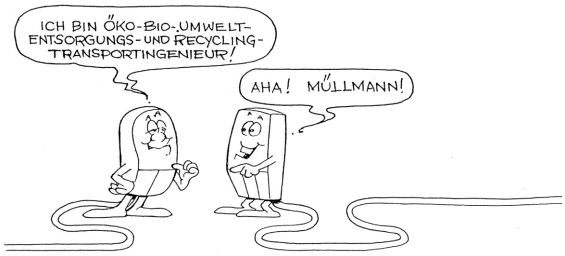
$$Dokumentation \equiv \left\{ \begin{array}{c} Konstruktions-Dokumentation \\ Konstruktions-Prozessdokumentation \end{array} \right.$$

In der Praxis entstehen Dokumente mehr oder weniger iterativ. Ideenskizzen werden ergänzt, verworfen, weiterentwickelt. Bei diesem (Iterations-)Prozess ist die allgemein übliche Arbeitsteilung in „so-eben-malskizziert" und in „Reinschrift-mit-ausreichender-Erklärung" auf kreative Ausnahmefälle zu begrenzen. Jede Aktivität im Rahmen des Konstruktionsprozesses ist gleich so zu notieren, dass sie Teil der Dokumentation sein kann. Unstrittig ist es ein echtes Problem, wie die entscheidenden Konstruktions-Absprachen, vielleicht notiert auf einem Bierdeckel, ohne Fehler in die Konstruktions-Prozessdokumentation gelangen (zumal keiner den Mut haben wird, den Bierdeckel dort abzuheften). Zu fordern ist daher, jede Idee unverzüglich rechnergestützt zu notieren, entweder direkt in ein LISP-System oder in einen Texteditor.

Sowohl in den ersten Phasen des Konstruktions-Prozesses als auch in den Phasen Wartung/Pflege und Fortentwicklung hat man das Problem, sich auf die wesentlichen Aussagen zu konzentrieren. Im ersten Fall fehlt eine stabile Basis (Spezifikation), im Zweiten sucht und vergleicht man und kann die Informationsfülle nicht verarbeiten. Beide Situationen zeigen, dass es notwendig ist, unsere Aussagen in einer auf das Wesentliche verkürzten Form zu notieren. Was wesentlich ist richtet sich:

1. nach den Adressaten und
2. dem Zweck des Dokumentes.

Adressaten-Ausrichtung:
Die einzelnen Dokumente wenden sich an unterschiedliche Adressaten, z. B. an Benutzer, Operateure, Auftraggeber, Projektmanager, Systemdesigner, Programmierer, Vertriebspersonal etc. Kaum jemand von ihnen liest eine Dokumentation gern, freiwillig oder mehr als nötig. Schon aus diesem Grunde ist die Dokumentation in „bewältigbare“ Abschnitte zu gliedern, die von den einzelnen Adressaten tatsächlich gelesen werden.



Wir formulieren keine Zeile ohne Annahmen über das Vorwissen und die vertraute Terminologie des Lesers. Zumeist unterstellen wir implizit ein unscharf durchdachtes Adressatenmodell. Notwendig ist jedoch hinreichende Klarheit über das annehmbare Leserverständnis. Welche Aspekte zu beschreiben sind und welche unerwähnt bleiben können, weil sie zum angenommenen „Leserfundus“ gehören, können wir nur anhand unseres unterstellten *Normadressaten* entscheiden.

Ist der Leser z. B. ein LISP-Programmierer, der den LISP-Quellcodetext zu pflegen hat, dann ist anzunehmen, dass dieser Adressat, die eingebauten, primitiven LISP-Konstrukte erkennt und versteht. Eine Kommentarzeile: *;;;Hier wird die Datei X geöffnet.* nach einem `open`-Konstrukt ist daher unsinnig. Sinnvoll wäre z. B. die Erläuterung: *;;;X ist als Datei abgebildet, damit der maximale Arbeitsspeicherbedarf kleiner als 1 MB bleibt*.

Da das Verständnis und der Kenntnisstand des Lesers mit der Benutzung der Dokumention wächst und sich infolgedessen seine Terminologie verändert, wäre eine Anpassung der Dokumentation an diese Veränderung wünschenswert. Soweit das Dokument rechnergestützt geführt wird, ist die Einprogrammierung verschieden kundiger Leserklassen (z. B. Anfänger, Geübter, Experte) ein üblicher Ansatz. Anzustreben ist eine Aktivitätsverschiebung. An die Stelle eines Dokumentes, das gelesen werden kann, tritt ein Dokumentations-Programm, das den Inhalt zugeschnitten auf den aktuellen Wissenstand des Lesers präsentiert. Lösungsansatz ist ein Dialog mit dem Leser, der Aufschluss über sein aktuelles Verständnis gibt (ohne dabei die Datenschutzaspekte zu missach-

ten!). Diese dynamische Klassifikation des Lesers wird genutzt, um eine Anpassung der anzuzeigenden Formulierungen zu realisieren.

Während der Adressat *Rechner* den beherrschten Sprachschatz eindeutig in Form der verarbeiteten Konstrukte und Konstruktionen offenbart, ist beim jeweiligen Leser ungewiss, ob dieser eine Formulierung tatsächlich hinreichend genau versteht. Kennt z. B. der Programmierer den Begriff `Rechnungsabgrenzungs-Konto`? Assoziiert er damit die Definition, die an anderer Stelle (vielleicht außerhalb der Dokumentationskette) in der Terminologie der Kaufleute formuliert wurde oder muss dieser Begriff in dem betreffenden Dokument erst erklärt werden oder genügt ein Verweis auf eine einschlägige Fundstelle?

Jede einzelne Formulierung hat sich an der Antwort auf die Frage zu orientieren: Welche Assoziationen kann der Adressat beim Lesen haben, welche davon sind nicht präzise genug oder stehen sogar im Widerspruch zum Gemeinten? Es geht um die Entscheidung: *Was ist unbedingt zu beschreiben, und was kann für sich selbst sprechen?* Je mehr für sich selbst spricht, um so kürzer wird das Dokument. Die Umfangreduktion ist im Hinblick auf die Lesemotivation ein positives Ziel. Wir wollen daher nur etwas (zusätzlich) erläutern, wenn der Leser ohne diese Erläuterung die Aussage nicht oder falsch versteht. Damit beugen wir einer kostenintensiven, unnützen und zum Teil schädlichen Aufblähung der Dokumentation vor.

Prinzipiell ist aus dem Quellcodetext (mit Kenntnis des jeweiligen LISP-Systems) die Abarbeitung nachvollziehbar. Das *Wie* ist, wenn auch recht mühsam, rekonstruierbar. Aus welchen Gründen wir ein Konstrukt gerade so und nicht anders definieren, ist vom Leser nicht ohne weiteres rekonstruierbar. Ein paar Sätze zur *Warum*-Darstellung sind daher oftmals hilfreicher als ausführliche Erläuterungen zum *Wie*. Wegen der Nicht-Rekonstruierbarkeit besteht eine besondere Pflicht, das *Warum* explizit zu notieren.

Zweck-Ausrichtung:
Ein Dokument ist auf die Zwecke auszurichten, die es erfüllen soll. Gliederung, Inhalt und Gestaltung (Layout) des Dokuments sind zweckorientiert zu erarbeiten. Ein Pflichtenheft, als eine Vertragsgrundlage zwischen Auftraggeber und Softwarekonstrukteur wird so gestaltet, dass Inhaltsänderungen nur über schriftliche Zusatzvereinbarungen möglich sind (d. h. z. B. gebundene Seiten mit durchlaufender Seitennummerierung gemäß *Seite i von n Seiten*). Ein Benutzerhandbuch ist stattdessen eine Loseblattsammlung, um Ergänzungen und Korrekturen an den richtigen Platz durch Austausch der Seiten aufnehmen zu können.

Ein Formulierungskonflikt besteht zwischen der Zweckausrichtung und dem Baukasten-Konstruktionsprinzip (Mehrfachverwendbarkeit). Die Aussagen sind so zu formulieren, dass eine Mehrfachverwendung

einzelner Passagen und/oder ganzer (Teil-)Abschnitte in anderen Dokumenten möglich ist. Beispielsweise sollen Textpassagen des Pflichtenheftes auch für Vertriebs- und Einführungs-Dokumente verwendbar sein. Diese Forderung nach Mehrfachverwendung verlangt möglichst kontextunabhängig zu formulieren. Die Zweckausrichtung führt jedoch zur Kontextintegration.

Zu vermeiden ist eine Kontextabhängigkeit, die sich nicht aus der Zweckausrichtung ableitet. Für ein Pflichtenheft sind daher z. B. die Aussagen nicht in der Zukunftsform zu notieren, sondern im Präsens. Weil es zunächst um etwas Zukünftiges geht, wählt der Autor spontan die Zukunftsform („Das Programm `Rechnung` wird die Mehrwertsteuer berechnen."). Erst die Notation im Präsens bringt die Herauslösung aus dem Kontext zukünftige Eigenschaft. („Das Programm `Rechnung` berechnet die Mehrwertsteuer.").

Für große Konstruktionen (↪ Tabelle 3.10 S. 406) sind Dokumentationsregeln verbindlich vorzuschreiben. Im Bereich der öffentlichen Verwaltung sind es z. B. die KoopA-ADV-Rahmenrichtlinien (Näheres dazu ↪ z. B. [22]). Solche Richtlinien bestimmen welchen Inhalt die einzelnen Dokumente des Konstruktionsprozesses haben (sollen!). Sie ermöglichen erst eine effiziente Kooperationen bei einer größeren Anzahl von Beteiligten und Betroffenen. Allerdings motiviert eine Richtlinie zum Schreiben, bei dem ein schnelles Abhaken einer lästigen Vorgabe in den Mittelpunkt rückt. Die angestrebte Vollständigkeit wird in der Praxis mit mehr oder weniger aussageleeren Phrasen erfüllt. Die pragmatische Empfehlung lautet daher: *Sei kein Formalist, der Richtlinien abhakt, sondern schreibe adressaten- und zweck-ausgerichtet!*

Im folgenden wird die Transparenz des Quellcodetextes behandelt. Dazu erörtern wir die drei Dokumentationsprobleme:

1. Benennung von Konstrukten,
2. Kommentierung und
3. Vor- und Rückwärtsverweise.

Diese Punkte sind von praktischer Bedeutung, wenn eine vorgegebene Spezifikation umzusetzen ist. Empfehlungen zur Erarbeitung einer solchen Spezifikation behandeln wir anschließend (↪ Abschnitt 3.2 S. 403).

3.1.1 Benennung

Der *Name* eines Konstruktes sei nicht irgendein Symbol, sondern benennt diejenige Eigenschaft, die es von allen anderen unterscheidet. Offensichtlich ist der Name bedeutsam für die Transparenz der gesamten

Dokumentation (incl. Quellcodetexte). Mit der Namensvergabe für „Objekte“, Funktionen, Makros, Variablen, Konstanten etc. sind zumindest zwei Aspekte verknüpft:

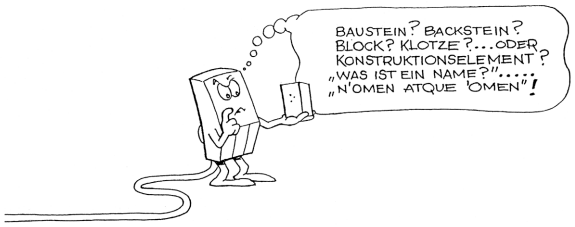
1. mnemotechnischer Aspekt

   Ziel ist eine leicht einprägsame Benennung im Sinne eines Verweises auf das „Gemeinte“ (Objekt) in der „realen“ Welt. Es geht um eine Korrelation zwischen der Bezeichnung des „realen“ Objektes und der Bezeichnung des „formalen“ Objektes (↪ Abschnitt 1.2.1 S. 52)

2. konstruktionsbedingter Aspekt

   Ziel sind Aussagen zu konstruktionsspezifischen Aufgaben und Eigenschaften (↪ Abschnitt 2.1 S. 141); z. B. Informationen zur Beantwortung der Fragen:

   - Ist es ein eingebautes Konstrukt?
   - Ist es ein risikoreiches Konstrukt?
   - ...



Die Benennung ist zusätzlich abhängig vom jeweiligen Denkrahmen (Paradigma) des Konstrukteurs (↪ Abschnitt 2.1 S. 141). Daher wird beispielsweise empfohlen, Funktionsnamen so zu wählen, „dass sie implizit Annahmen über ihre Arbeit in Form von ≫Verträgen≪ über Eingabe-Ausgabe-Relationen aufdrängen.“ (↪ [178], S. 56). Im objektgeprägten Denkrahmen sind Operationen mit Verben und Operanden mit Substantiven zu benennen. Im imperativ-geprägten Denkrahmen haben alle Bezeichnungen Befehlscharakter an den ausführenden Rechner („Kasernenhof“-Formulierungen). Orientiert am jeweiligen Denkrahmen wären für ein Konstrukt, das z. B. den Firmenkurznamen eines Kunden selektiert, folgende Benennungen zu wählen:

- Imperativ-geprägte Konstruktion:
  `Gib-Firmen-Kurzname`
- Funktions-geprägte Konstruktion:
  `Kunde->Firmen-Kurzname`

- Objekt-geprägte Konstruktion:
  `Selektieren-Firmen-Kurzname`

Als konstruktionsbedingte Informationen wollen wir mit der Benennung z. B. vermitteln:

1. den (Daten-)Typ

   Ist es eine Funktion, eine Variable, ein Makro etc.? Z. B. :
   ```
   F-Bilanzabgleich   ;Funktion
   V-Bilanzabgleich   ;Variable
   M-Bilanzabgleich   ;Makro
   ```

2. den aufgabensspezifischen Typ

   Ist es ein Konstruktor, Prädikat, Selektor oder Mutator? (↪ Tabelle 3.2 S. 380)

3. die Flussrichtung

   In welche Richtung fließen die Daten? Z. B. :
   `Input-Kundendatei` oder `Output-Kundendatei`

4. den Wertebereich

   Ist es ein Zahlatom oder String? Z. B. :
   `Zahl-Bilanz-Aktiva` oder `String-Bilanz-Aktiva`

5. die Aufgabe

   Ist es eine `lambda`-Variable, ein Argument, eine Hilfsvariable etc.? Z. B. :
   ```
   Bilanz_Aktiva   ;Unterstrich für eine
                   ; lambda-Variable
   Bilanz-Aktiva   ;Bindestrich für Argument
   BA              ;weniger als 3 Buchstaben für
                   ; temporäre Hilfsvariable
   ```

6. die Lebensdauer (Persistenz)

   Ist der Speicherort eine Stamm-, Bewegungs-, Hilfs-Datei oder der Arbeitsspeicher? Z. B. :
   ```
   DB-Kunde   ;Stammdatei (Datenbank)
   WS-Kunde   ;Arbeitsspeicher
              ; (working storage)
   ```

Empfehlungen zur Namensvergabe beziehen sich häufig auf konstruktionsspezifische Informationen. So wird empfohlen, einen Konstruktor

Bewertungskriterium	langer Name	kurzer Name
(mnemotechnische) Aussagekraft	+	–
Lesbarkeit/Leseaufwand	–	+
Schreibaufwand	–	+
Schreibfehlerrisiko	–	+

Legende:
+ ≡ Vorteil
– ≡ Nachteil

Tabelle 3.1: Namenslänge —- Vor- und Nachteile

$$< name > \equiv \{ < praefix >\} < stamm > \{ < suffix > \}$$

Empfehlung:				Beispiel:
für einen Konstruktor:	`Make-`	≡	$< praefix >$	`Make-Window`
für ein Prädikat:	`?`	≡	$< suffix >$	`EOF-Object?`
für ein destruktives Konstrukt (Mutator):	`!`	≡	$< suffix >$	`set-cdr!`
für eine besondere (globale) Variable:	`*`	≡	$< praefix >$	
	`*`	≡	$< suffix >$	`*GRAPHICS-COLORS*`
für eine eingebaute Konstante:	`#`	≡	$< praefix >$	`#!eof`

Tabelle 3.2: *Affix*-Empfehlungen

oder Selektor derart zu bezeichnen, dass man den Datentyp ersehen kann, den die Schnittstelle des Konstruktes erwartet. Ein Beispiel für einen solchen Namen ist: `ARGLIST!FIRST-ARG` (↪ [173], S.176). Oder es wird präzisiert, in welchen Fällen die Selektoraufgabe vor der Prädikataufgabe im Namen Vorrang hat. In Scheme haben daher die Konstrukte `memq`, `memv`, `member`, `assq`, `assv` und `assoc` kein Fragezeichen am Namensende, weil sie nützliche Werte und nicht nur den Standardwahrheitswert `#t` zurückgeben (↪ [151]).

Sind viele Aspekte mit dem Namen zu vermitteln, dann führt dies zwangsläufig zu sehr langen Namen. Lange Namen vergrößern den Umfang der Dokumente. Sie führen darüber hinaus schnell zu Problemen bei einer begrenzten Zeilenlänge. Vor- und Nachteile bezogen auf die Namenslänge zeigt Tabelle 3.1 S. 380.

Ein Name, insbesondere ein langer Name, ist zu strukturieren. Wir unterteilen einen Namen in einen Vorspann (*Präfix*), den Hauptteil (*Stamm*) und einen Nachspann (*Suffix*). Die Tabelle 3.2 S. 380 zeigt die verwendete Regelung für *Präfix* und *Suffix*, die zusammen als *Affix* bezeichnet werden.

Die Optionen zur Wahl und Strukturierung eines zweckmäßigen Namens erörtern wir anhand eines Beispiels. Es sei der Kurzname der Firma aus den Kundendaten zu selektieren. Dazu betrachten wir sowohl die Namen bei der Funktionsdefinition, also die Benennung der `lambda`-Abstraktion, als auch die Funktionsapplikation, also zusätzlich die Namen der Argumente. Uns interessieren die Wirkungen der unterschiedlichen Benennung im Hinblick auf die Transparenz der gesamten Konstruktion. Dazu betrachten wir exemplarisch vier Entscheidungsprobleme:

1. die Informationsaufteilung zwischen dem Funktionsnamen und den Namen für die `lambda`-Variablen,

2. das (standardisierte) Einbeziehen von konstruktionsbedingten Informationen,

3. die Informationsaufteilung zwischen Namen und einem zugeordnetem (Daten-)Lexikon und

4. die Sortierfähigkeit für ein schnelles Recherchieren.

Aus Vereinfachungsgründen liegen die Kundendaten in unserem Beispiel als Liste vor, wobei der Firmenkurzname als optionales zweites Listenelement angegeben ist. Zwei Kunden sind vorab folgendermaßen definiert:

```
eval> (define Kunde-1 '("Software AG" "SAG"))
eval> (define Kunde-2 '("Siemens AG"))
```

Ist der Firmenkurzname nicht angegeben, dann ist ein Ersatzwert zu selektieren. Dieser ist, wie folgt, definiert:

```
eval> (define *UNBEKANNTER-KURZNAME* "-?-")
```

Problem 1: Informationsaufteilung zwischen Funktionsname und Namen der `lambda`-Variablen

Bei der Benennung des gewünschten Selektors ist zu entscheiden, welche Informationen im Funktionsnamen und welche in den Namen der `lambda`-Variablen zu hinterlegen sind.

Lösung A:

- Funktionsname Haupt-Informationsträger
- Kurze beliebige Namen für die `lambda`-Variablen

```
eval> (define Kunden->Firmen-Kurzname/Ersatz
        (lambda (X . Y)
          (cond ((= (length X) 1)
                 (cond ((= (length Y) 1)
                        (list-ref Y 0))
                       (#t *UNBEKANNTER-KURZNAME*)))
                (#t (list-ref X 1)))))

eval> (Kunden->Firmen-Kurzname/Ersatz
        Kunde-1 *UNBEKANNTER-KURZNAME*)
       ==> "SAG"
```

Lösung B:

- Funktionsname und Namen der `lambda`-Variablen sind gleichbeteiligte Informationsträger

```
eval> (define Firmen-Kurzname
        (lambda (Kunde . Ersatz_Kurzname)
          (cond ((= (length Kunde) 1)
                 (cond ((= (length Ersatz_Kurzname) 1)
                        (list-ref Ersatz_Kurzname 0))
                       (#t *UNBEKANNTER-KURZNAME*)))
                (#t (list-ref Kunde 1)))))

eval> (Firmen-Kurzname Kunde-1 *UNBEKANNTER-KURZNAME*)
      ==> "SAG"
```

Problem 2: Einbeziehen von konstruktionsbedingten Informationen

Die Namen in den Lösungen A und B enthalten keine Typ-Aussagen. So ist nicht erkennbar, dass die Funktion ein Selektor ist und ihr erstes Argument eine Liste sein muss.

Lösung C:

- Präfix des Funktionsnamens verweist auf den Typ Selektor
- `lambda`-Variablen geben Informationen in Bezug auf die zulässigen Argumente

```
eval> (define Selektiere-Firmen-Kurzname
        (lambda (Liste . Ersatz_String)
          (cond ((= (length Liste) 1)
                 (cond ((= (length Ersatz_String) 1)
                        (list-ref Ersatz_String 0))
                       (#t *UNBEKANNTER-KURZNAME*)))
                (#t (list-ref Liste 1)))))

eval> (Selektiere-Firmen-Kurzname
        Kunde-2 *UNBEKANNTER-KURZNAME*)
      ==> "-?-"
```

Diese Lösung ist auch weniger imperativ-geprägt, wie folgt, formulierbar:

```
eval> (define Selektion-Firmen-Kurzname
        (lambda (Liste . Ersatz_String)
          (cond ((= (length Liste) 1)
                 (cond ((= (length Ersatz_String) 1)
                        (list-ref Ersatz_String 0))
                       (#t *UNBEKANNTER-KURZNAME*)))
                (#t (list-ref Liste 1)))))
eval> (Selektion-Firmen-Kurzname
        Kunde-2 *UNBEKANNTER-KURZNAME*)
      ==> "-?-"
```

Die Information, dass die Funktion einen Selektor darstellt, ist sicherlich kürzer mit einem vorgegebenen Präfix, z. B. `get-`, notierbar. Die Information, dass die erste `lambda`-Variable an eine Liste zu binden ist, kann ebenfalls kürzer notiert werden. Wir können für eine ganze Konstruktion vereinbaren, dass z. B. eine `lambda`-Variable mit dem Namen `L`, stets ein Argument vom Typ Liste erfordert. Die zweite `lambda`-Variable im Beispiel verweist auf einen String. Diese Information ist in diesem Funktionskörper unerheblich. Die `lambda`-Variable `Ersatz_String` wird aufgrund der Punkt-Paar-Notierung in der Schnittstelle des Konstruktes stets an eine Liste gebunden (↪ Abschnitt 2.3.3 S. 222). Das Konstrukt `(list-ref Ersatz_String 0)` selektiert das erste Element, unabhängig davon, von welchem Typ es ist. Wir können daher in diesem Konstrukt auf den Namensteil `String` verzichten.

Lösung D:

- Standard-Selektor-Präfix beim Funktionsnamen
- Standard-Namen für `lambda`-Variablen mit Informationen in Bezug auf die zulässigen Argumente (soweit erforderlich)

```
eval> (define Get-Firmen-Kurzname
        (lambda (L . Ersatz)
          (cond ((= (length L) 1)
                 (cond ((= (length Ersatz) 1)
                        (list-ref Ersatz 0))
                       (#t *UNBEKANNTER-KURZNAME*)))
                (#t (list-ref L 1)))))

eval> (Get-Firmen-Kurzname
        Kunde-2 *UNBEKANNTER-KURZNAME*)
      ==> "-?-"
```

Problem 3: Informationsaufteilung zwischen Namen und Lexikon

Nachteilig ist, insbesondere bei häufiger Applikation der Lösung D, dass der Name `Get-Firmen-Kurzname` mit 19 Zeichen recht lang ist. Wir verkürzen ihn daher. Zusätzlich präzisieren wir bei der `lambda`-Variable `Ersatz`, wofür diese als Ersatz dienen soll. Wir nennen sie jetzt `Name` und verweisen mit dem Sonderzeichen `?` als Präfix auf ihre Aufgabe. Sie ist Ersatz für den Firmenkurznamen, wenn dieser in den Kundendaten unbekannt ist.

Lösung E:

- Verkürzung der Namen von Lösung D
- Nutzung von Sonderzeichen

```
eval> (define Get-F-K-Name
        (lambda (L . ?_Name)
          (cond ((= (length L) 1)
                 (cond ((= (length ?_Name) 1)
                        (list-ref ?_Name 0))
                       (#t *UNBEKANNTER-KURZNAME*)))
                (#t (list-ref L 1)))))

eval> (Get-F-K-Name
        Kunde-2 *UNBEKANNTER-KURZNAME*)
      ==> "-?-"
```

Dieses Verkürzen führt rasch zu Namen, deren Bedeutung sich niemand merken kann. Wofür steht `F` und wofür `K` ? In der Praxis ist dann permanent in einem (Daten-)Lexikon nach der eigentlichen Bedeutung nachzuschauen.

Das Suchen können wir für das rechnergestützte Lesen vereinfachen. Mit Hilfe der Eigenschaftsliste speichern wir für jeden verkürzten Namen die vollständige Fassung. Dazu formulieren wir zwei Makros. Der Makro `::=` schreibt die Erläuterung in die Eigenschaftsliste. Der Makro

`?` ermöglicht das Nachfragen. Da die Eigenschaftsliste global ist, kann in jeder Umgebung nachgefragt werden (zu den Nachteilen dieser Globalität ↪ Abschnitt 2.2.3 S. 191).

Lösung F:

- Lösung E in Verbindung mit einem (Daten-)Lexikon als Eigenschaftsliste (P-Liste ↪ Abschnitt 2.2.3 S. 191).

Das `gensym`-Konstrukt gewährleistet, das keine andere Eigenschaft überschrieben wird (↪ Abschnitt 2.4.4 S. 259).

```
eval> (define *IST* (gensym))
eval> (define-syntax-rule (::= Symbol Doku)
        (putprop Symbol *IST* Doku))
eval> (define-syntax-rule (? Symbol)
        (getprop Symbol *IST*))
eval> (define Get-F-K-Name
         (begin
           (::= 'Get-F-K-Name
             "Kurzname der Firma")
           (::= '?_Name
           "Ersatzwert für den Kurznamen der Firma")
           (lambda (L . ?_Name)
             (cond ((= (length L) 1)
                    (cond ((= (length ?_Name) 1)
                           (list-ref ?_Name 0))
                          (#t
                           *UNBEKANNTER-KURZNAME*)))
                   (#t (list-ref L 1)))))

eval> (Get-F-K-Name Kunde-1 *UNBEKANNTER-KURZNAME*)
      ==> "SAG"
eval> (? 'Get-F-K-Name)
     ==> "Kurzname der Firma"
eval> (? '?_Name)
    ==> "Ersatzwert für den Kurznamen der Firma"
```

Das so mitgeführte Lexikon ist eine wenig effiziente, beschränkt nützliche Lösung, die beim Testen hilft (↪ auch *documentation string*, Abschnitt 3.1.2 S. 389). In allen nicht rechnergestützten Lesefällen ist sie nicht besser als eine Semikolon-Notation (↪ Abschnitt 3.1.2 S. 389).

Problem: Reihenfolge im Lexikon

Mit dem Verweis auf eine zusätzliche Beschreibung in einem Lexikon wird deutlich, dass der Name so zu wählen ist, dass wir diese Beschreibung leicht finden; auch im Suchfall ohne direkte Rechnerunterstützung.

Mit einem standardisierten Präfix für Selektoren (hier: `Get-`) werden diese im Lexikon alle hintereinander ausgewiesen. Wahrscheinlich bevorzugen wir jedoch eine Ordnung, bei der unser Verbund von Konstruktor, Selektor, Prädikat und Mutator (↪ Abschnitt 2.1.2 S. 146) hintereinander ausgewiesen wird. Suchen wir z. B. die Beschreibung für `window`-Konstrukte in einer LISP-Bibliothek, dann durchlaufen wir folgende Reihenfolge der Lexikoneinträge:

```
window-clear
window-delete
window-get-attribute

⋮

window-set-size!
window?
```

Selbst hier ist häufig, z. B. bei *PC Scheme*, der zugehörende Konstruktor nicht konsequent mit `window-make`, sondern mit `make-window` bezeichnet und befindet sich daher viel weiter vorn im Lexikon.

Wir können die gewünschte Reihenfolge erreichen, indem wir jeweils einen Standard-Suffix für Konstruktor, Selektor, Prädikat und Mutator verwenden.

Lösung G:

- Standard-Suffix für den Verbund von Konstruktor, Selektor, Prädikat und Mutator

```
eval> (define Firmen-Name-Make
        (lambda (Langname . Kurzname)
          (if Kurzname (cons Langname Kurzname)
                       (list Langname))))

eval> (define Kunde-1
        (Firmen-Name-Make "Software AG" "SAG"))
eval> (define Kunde-2
        (Firmen-Name-Make "Siemens AG"))

eval> (define Firmen-Kurzname?
        (lambda (Kurzname)
          (and (string? Kurzname)
               (<= (string-length Kurzname) 3))))
eval> (define Firmen-Kurzname-Get
        (lambda (L . Ersatz)
          (cond ((= (length L) 1)
                 (cond ((= (length Ersatz) 1)
                        (list-ref Ersatz 0))
```

```
                        (#t *UNBEKANNTER-KURZNAME*)))
                 (#t (list-ref L 1)))))
eval> (define-syntax-rule
        (Firmen-Kurzname-set! L Kurzname)
          (set! L (cons (car L)
                        (list Kurzname))))

eval> (Firmen-Kurzname?
        (Firmen-Kurzname-Get Kunde-1))
      ==> #t
eval> (Firmen-Kurzname-Get Kunde-2)
      ==> "-?-"
eval> (Firmen-Kurzname-set! Kunde-2 "SIE")
eval> (Firmen-Kurzname-Get Kunde-2)
      ==>"SIE"
```

Die Berücksichtigung der Reihenfolge führt zu Formulierungen, die im Vergleich zur Präfix-Verschlüsselung (Lösung D), ungewohnt und weniger flüssig zu lesen sind.

```
eval> Firmen-Kurzname-Get ==> ...   ;zweckmäßige
                                    ; Sortierbarkeit
eval> Get-Firmen-Kurzname ==> ...   ;relativ gute
                                    ; Lesbarkeit
```

Wir haben für die Strukturierung eines Namens die Sonderzeichen, Bindestrich und Unterstrich, verwendet. Unsere Konvention reserviert den Bindestrich für Funktionsnamen und damit auch für Argumente. Der Unterstrich ist für die `lambda`-Variablen reserviert.

Option: Groß/Kleinschreibung

Eine Strukturierung sollte aufgrund der Groß/Kleinschreibung erfolgen. Dabei nutzen wir dann den Vorteil, dass Geschriebenes in Kleinbuchstaben um $\approx 20 \ldots 50$ Prozent schneller als in Großbuchstaben gelesen wird (untersucht von Stuart/Stuart Gardner, 1955; zitiert nach [35]). Wesentliche Ursache ist die Wellenform der oberen und unteren Linie bei Wörtern mit Kleinbuchstaben; z. B. `Der neue Weg` statt `DER NEUE WEG`. Die Groß/Kleinschreibung sollte heute stets genutzt werden.[3]

Lösung H:

- Alle Zeichen mit Fluchtsymbol zur Namensstrukturierung

```
eval> (define |Selektion Firma Kurzname|
        (lambda (Liste . |Ersatz String|)
```

[3] Die ursprünglichen LISP-Systeme konnten nur mit Großbuchstaben arbeiten und konvertierten daher in der *READ*-Phase Kleinbuchstaben in Große (↪ Abschnitt 2.4.3 S. 248).

```
          (cond ((= (length Liste) 1)
                 (cond ((= (length |Ersatz String|) 1)
                        (list-ref |Ersatz String| 0))
                       (#t *UNBEKANNTER-KURZNAME*)))
                (#t (list-ref Liste 1)))))
eval> (|Selektion Firma Kurzname|
        Kunde-2)
      ==> "SIE"
```

Mit dem (mühsam)[4] notierten Fluchtsymbol kann man Leerzeichen zum Strukturieren des Namens verwenden. Ein Name könnte somit ein ganzer Satz sein oder auch nur eine Zahl.

```
eval> (define
        |Die Reppenstedter Modewelt ist unbedeutend.|
         (Firmen-Name-Make "Mode GmbH" "MoG"))
eval> |Die Reppenstedter Modewelt ist unbedeutend.|
      ==> ("Mode GmbH" "MoG")
eval> (define |4711|
       (Firmen-Name-Make "Köln Wasser AG" "KWG"))
```

Option: *Alias*-**Namen**

Mit Hilfe des `alias`-Konstruktes kann ein Name wie ein anderer, schon existierender Name verwendet werden. Beide Namen bezeichnen dieselbe „Einheit“. Man könnte spontan einen *Sonntagsnamen* und einen *Alltagsnamen* kreieren. Der *Sonntagsname* ist lang und enthält die gewünschten Informationen. Der *Alltagsname* ist eine Kurzschreibweise dessen. Zunächst wird nur der *Sonntagsname* genutzt. Käme dieser Name jedoch in einem Textbereich häufig vor, dann wechselt man zum *Alltagsnamen*. Man mutet dem Leser zu, sich die Bedeutung eines häufig notierten Namens einzuprägen, um die Verkürzungsvorteile nutzen zu können.

Lösung I:

- zwei Namen

```
eval> (define-syntax-rule (--> Symbol Wert)
        (define Symbol Wert))
eval> (define-syntax-rule (f x)
        (eval `(lambda ,@x)))

eval> (--> Firmen-Kurzname-Get
           (f '((L . Ersatz)
              (cond ((= (length L) 1)
```

[4] LISP-angepasste Editoren erleichtern die Eingabe des Fluchtsymbols, indem sie gleich zwei senkrechte Striche mit der aktuellen Cursor-Stellung in der Mitte bei Eingabe eines senkrechten Striches erzeugen.

```
                      (cond ((= (length Ersatz) 1)
                             (list-ref Ersatz 0))
                            (#t
                             *UNBEKANNTER-KURZNAME*)))
                     (#t (list-ref L 1))))))
eval> (Firmen-Kurzname-Get Kunde-2)
      ==> "-?-"
```

Die Gefahren, die eine solche doppelte Benennung hervorruft, sind nicht zu unterschätzen. Im Analysefall sind stets beide Namen zu verfolgen. Wir wählen daher für selbstdefinierte Mutatoren stets nur einen Namen.

Beachten wir bei der Wahl eines Namens nicht die Informationsaufteilung mit dem (Daten-)Lexikon und mit weiteren Namen, die z. B. im Zusammenhang mit seiner Anwendung relevant sind, dann wird der Name lang, und wir müssen die damit verbundenen Nachteile (↪ Tabelle 3.1 S. 380) in Kauf nehmen (Lösung A ↪ S. 382). Anzustreben ist eine Rechnerunterstützung beim Lesevorgang, weil damit ein zweckmäßiger Kompromiss zwischen Namen und Lexikon erreichbar ist (Ansatz z. B. Lösung F ↪ S. 385). Die Groß/Kleinschreibung ist eine geeignete Strukturierungsoption für den Namen, vorausgesetzt, es bedarf keiner Fluchtsymbolnotation. Zur Vermittlung konstruktionsbedingter Aspekte ist im Hinblick auf kurze Namen die Verwendung normierter Affixe zweckmäßig (Lösungen D ↪ S. 383 und G ↪ S. 386).

3.1.2 Kommentierung

Einerseits wissen wir (zumindest) seit der klassischen Diskussion über die strukturierte Programmierung („*Go To Statement Considered Harmful*“; Dijkstra, 1968 ↪ [50]), dass ein undurchschaubarer Quellcodetext durch eingefügte Kommentarzeilen kaum transparenter wird. Springen wir kreuz und quer (d. h. es liegt eine nichtlineare Kontrollstruktur vor), dann kann der Transparenzmangel nicht mit einer Kommentierung (quasi nebenläufig) behoben werden. Andererseits wissen wir, dass sehr komplexe Konstruktionen mit unstrukturierten Konstrukten relativ kurz notiert werden können. Zu Recht sind daher auch Strukturen zu berücksichtigen, „*die wir heute als 'ziemlich unstrukturiert' empfinden und vor denen wir in der Ausbildung (noch) warnen*“ (↪ [38], S. 50). Beispielsweise sind wir mit dem `call/cc`-Konstrukt (↪ Abschnitt 2.5.2 S. 285) in der Lage komplizierte nichtlineare Strukturen abzubilden.

Zur Transparenz solcher nichtlinearen Strukturen können verbale Kommentarzeilen nur begrenzt beitragen. Letztlich helfen sie nur bei transparenten Programmen. Diese brauchen Kommentare jedoch nicht, da sie ja schon transparent sind; im Gegenteil, eine Kommentierung verlängert das Dokument, erhöht den Wartungsaufwand, weil die Kommentare (mit)gepflegt werden müssen und schadet daher. So gesehen

müssten wir jede Kommentarzeile ablehnen. Vorsichtshalber wäre eine Streichung des Semikolons als Kommentarpräfix aus unserem LISP-System vorzunehmen (↪ [102]).

Im Gegensatz dazu setzen viele Dokumentationsempfehlungen auf die intensive Kommentierung: „*Großzügiger Gebrauch von Kommentaren ist guter Programmierstil*“ (↪ [190], deutsche Fassung S. 48) oder ”*. . ., I can't recommend them highly enough*” (↪ [78], p. 39). Wir befürworten einen pragmatischen Standpunkt und empfehlen, Kommentarzeilen sparsam zu verwenden, und warnen vor einer „Überkommentierung“ (↪ z. B. auch [107], S. 377). Ihre Aufgabe ist die *Warum*-Erläuterung und nur in Fällen, wenn unserem Adressaten die Abarbeitungsfolge fremd sein wird, auch die *Wie*-Erläuterung. So werden Kommentare kein Ersatz für eine unzulängliche Benennung von Konstrukten. Umgekehrt muss die Benennung nicht mit Informationen zum *Warum* belastet werden.

Nicht zu übersehen sind die gravierenden Nachteile von Kommentarzeilen. Zunächst ist zu vereinbaren, ob sie das ihnen nachfolgende Programm(fragment) beschreiben oder das vorhergehende. Selbst bei strikter Einhaltung einer entsprechenden Vereinbarung (↪ Tabelle 3.3 S. 391) bleibt der Nachteil, dass die Reichweite eines Kommentars nicht unmittelbar erkennbar ist, sondern durch Interpretation des Textes zu folgern ist.

Konstruktionsmäßig wird eine Kommentarzeile eher wie ein global definierter Wert behandelt. Die Kommentierung bleibt damit auf dem Erkenntnisniveau primitiver Programmiersprachen stehen, die keine Blockstrukturen, d. h. keine lokalen Werte, kennen. Wünschenswert wäre ein Mechanismus zur Kommentierung, der die Reichweite und die Verknüpfung mit dem jeweiligen Programmfragment gewährleistet (↪ [13]). Solange das eingesetzte LISP-System einen solchen Mechanismus nicht bietet, sind wir auf eine Ersatzlösung angewiesen. Eine einfache Lösung zeigt Tabelle 3.3 S. 391.

Insbesondere aus dem Umfeld vom *Massachusetts Institute of Technology Cambridge* (MIT[5]) stammt eine Konvention mit Semikolon-Folgen, die auch Eingang in die *Common LISP*-Beschreibung gefunden hat (↪ [174], S. 348).

Ein Semikolon, also *Einfach-Semikolon-Kommentar*, dient zur Kommentierung der Zeile, auf der es steht. Dabei sind mehrere Einfach-Semikolon-Kommentare spalten mäßig auszurichten, d. h. sie beginnen stets in der gleichen Spalte. Ist der Kommentar länger als die verbleibende Restzeile, dann wird er auf der nächsten Zeile fortgesetzt, allerdings folgt diesem Semikolon ein Leerzeichen.

Ein *Doppelter-Semikolon-Kommentar* beginnt in der Spalte, in der das zu erläuternde Konstrukt anfängt. Nach dem doppelten Semikolon folgt ein Leerzeichen. Die Kommentierung bezieht sich in der Regel auf das

[5] MIT ↪ `http://web.mit.edu/` (Zugriff: 3-Jan-2010)

Symbol	**Erläuterung**
$;<comment>$	Markiert die Stelle des Beginns der Reichweite eines Kommentars.
$;</comment>$	Markiert die Stelle des Endes der Reichweite eines Kommentars.
$;<!--comment-->$	Einzeiliger Kommentar, dessen Wirkung dokumentabhängig entweder das vorher- oder das nachfolgende Konstrukt erläutern.

Legende:
Gemäß *Extensible Markup Language* (XML), der Idee „strukturierte Daten“ jeglicher Art, z. B. auch Bilder, in Form von Textdateien zu verwalten und mit *Stylesheets* (Textdateien) auszuwerten.
XML-Standard: ↪ `http://www.w3.org/TR/xmlbase/` (Zugriff: 3-Jan-2010)

Tabelle 3.3: Kennzeichnung der Reichweite eines Kommentars

nachstehende Konstrukt.

Ein *Dreifach-Semikolon-Kommentar* beginnt stets in der ersten Spalte einer Zeile. Er beschreibt ein größeres Konstrukt. Ein *Vierfach-Semikolon-Kommentar* kennzeichnet Überschriften. Er beginnt ebenfalls in der ersten Spalte.

Die Beispiele in diesem Buch sind weitgehend gemäß dieser Konvention (↪ Tabelle 3.4 S. 392) kommentiert.

Kommentarzeilen sind gleichzeitig mit den LISP-Zeilen zu schreiben, die sie erläutern. Es ist sicherzustellen, dass Kommentare mitgepflegt werden. Ein nicht mehr gültiger Kommentar ist irreführend und daher schädlicher als ein fehlender Kommentar. Bei jeder Modifikation empfiehlt es sich, vorsichtshalber Kommentare in Bezug auf die Änderungsstelle vorab zu löschen.

Exkurs: *Documentation String*
In *Common LISP* kann beim Definieren einer Funktion (mit dem `DEFUN`-Konstrukt) oder eines Makros (mit dem `DEFMACRO`-Konstrukt) im Deklarationsbereich ein Kommentar hinterlegt werden:

$$\texttt{(DEFUN} < funktionsname > < lambda-schnittstelle > \{< deklaration >\} < funktionskoerper > \texttt{)}$$

Ist eine $<$ *deklaration* $>$ vom Typ `String`, dann dient sie zur Kommentierung des Konstruktes. Der Kommentar ist abrufbar mit der Applikation von `DOCUMENTATION`, z. B.:

```
eval> (DOCUMENTATION 'FOO 'FUNCTION)
```

Semikolonanzahl	Beginn in der Spalte	**Verwendung**
`;;;;{#\space}`	1	Überschrift für Kapitel, Abschnitte und Unterabschnitte
`;;;{#\space}`	1	Kommentar für ein gößeres Konstrukt (z. B. Funktion)
`;;{#\space}`	des Konstruktes	Kommentar für das (nachfolgende) Konstrukt
`;`	lokal vorgegeben	Kommentar für diese Zeile. Falls Platzmangel, dann nächste Zeile mit Semikolon, gefolgt von einem Leerzeichen.

Legende:
`{#\space}` ≡ Leerzeichen in den Folgezeilen
Ähnlicher Vorschlag in *Common LISP*; Quelle: ↪ [174]

Tabelle 3.4: Eine bewährte Konvention für Kommentare

```
          ==> "Kommentar-Text"
```

Das folgende `Array`-Beispiel zeigt eine Situation, bei der eine Kommentierung notwendig ist, obwohl die Kommentarzeile das *Wie* beschreibt.

Beispiel: `Array` **initialisieren** Es ist eine zweidimensionale Tabelle (engl.: *array*) zu konstruieren. Dazu sind die beide folgenden Konstrukte `Make-Array` und `Array-set!` definiert.

```
eval> (define Make-Array
        (lambda (Max_Zeilen Max_Spalten)
          (let ((V (make-vector Max_Zeilen)))
            (do ((i 0 (+ i 1)))
                ((> i (- Max_Zeilen 1)) V)
              (vector-set! V
                           i
                          (make-vector
                            Max_Spalten))))))

eval> (define Array-set!
        (lambda (Array Zeile Spalte Neuer_Wert)
          (let ((Neue-Zeile
                 (begin
                   (vector-set!
                    (vector-ref Array
                                (- Zeile 1))
                    (- Spalte 1)
                    Neuer_Wert)
                   (vector-ref Array
```

```
                                (- Zeile 1)))))
                (vector-set! Array
                             (- Zeile 1)
                             Neue-Zeile))))
```

Mit dem Konstrukt `Make-Array` definieren wir ein Tabellenbeispiel `Foo` mit 4 Zeilen und 5 Spalten. Das anschließende `let`-Konstrukt modifiziert den Wert von `Foo`. Wir beurteilen hier nicht, ob es sich um ein trickreiches Konstrukt handelt, das besser durch eine durchschaubare Lösung zu ersetzen wäre (dazu ↪ [106]). Uns interessiert das Bestehen der Kommentierungs-Pflicht.

```
eval> (define Foo (Make-Array 4 5))
eval> (let ((n 4) (m 5))
        (do ((i 1 (+ i 1)))
            ((> i n))
            (do ((j 1 (+ j 1)))
                ((> j m))
                (Array-set! Foo i j
                   (* (quotient i j)
                      (quotient j i))))))
eval> Foo ==>
#(#(1 0 0 0 0)
  #(0 1 0 0 0)
  #(0 0 1 0 0)
  #(0 0 0 1 0))
```

Der neue Wert von `Foo` ist für den angenommenen Leser nicht sofort nachvollziehbar. Dieser muss erkennen, dass das Argument `(*(quotient i j )(quotient j i))` nur die beiden Werte 0 und 1 annehmen kann. Ist `(eq? i j)` erfüllt, dann ist es der Wert 1, andernfalls ist es der Wert 0. Das `quotient`-Konstrukt hat als Rückgabewert die Integerzahl der Divisionsrechnung. Es ist daher adressatengerecht, auf diese Integerdivision hier mit einem Kommentar hinzuweisen.

```
eval> (let ((n 4) (m 5))
        (do ((i 1 (+ i 1)))
            ((> i n))
            (do ((j 1 (+ j 1)))
                ((> j m))
                (Array-set! Foo i j
                   ;;Integer-Division, Wert
                   ;; entweder 0 oder 1
                   (* (quotient i j)
                      (quotient j i))))))
```

3.1.3 Vorwärts- und Rückwärtsverweise

In den ersten Phasen des Konstruierens mag ein „lokal sequentielles“ Lesen und Vergleichen mit anderen Passagen im selben Dokument oder

Eigenschaften der Dokumentation:

1. eindeutig,
2. vollständig,
3. verifizierbar,
4. konsistent,
5. modifizierbar,
6. zurückführbar (traceable) und
7. handhabbar.

Tabelle 3.5: Anzustrebende Eigenschaften für eine Dokumentation

in anderen dominieren. Spätestens in der Wartungs- und Pflegephase herrscht das Lesen als Suchprozess vor. Über Verzeichnisse und Querverweise wird mehr oder weniger diagonal gelesen, um z. B. Fehler zu isolieren oder versteckte Abhängigkeiten aufzudecken.

Daher ist es sinnvoll, von Anfang an das Suchen in einer Dokumentation zu unterstützen. Vorwärts- und Rückwärtsverweise sind deshalb Bestandteil einer „guten“ Dokumentation. Tabelle 3.5 S. 394 nennt die anzustrebenden Eigenschaften für eine Dokumentation (↪ z. B. [6]).

Die Eigenschaft *Eindeutigkeit* bedingt, dass wir mehrdeutige Begriffe vermeiden oder zumindest ihre angenommene Bedeutung in einem Begriffslexikon (*Glossary*) hinterlegen. Die Eigenschaft *Vollständigkeit* ist adressaten- und zweckausgerichtet zu erfüllen. Ist der Adressat der Rechner, dann ist *Vollständigkeit* (in der Praxis nächträglich) objektiv feststellbar. In allen anderen Fällen ist subjektiv zu entscheiden, ob etwas explizit zu beschreiben ist oder nicht. Die Eigenschaften *Verifizierbarkeit* und *Konsistenz* (Widerspruchsfreiheit) sind, soweit es sich um verbale, nicht streng formaliserte Texte handelt, nicht exakt definierbar und kaum erreichbar. Sie bilden jedoch schon intuitiv einen Maßstab, der zur Präzision und zum Einbeziehen schon formulierter Aussagen zwingt.

Die Eigenschaft *Modifizierbarkeit* können wir entsprechend dem folgenden Prädikat `Modifizierbar?` weiter aufschlüsseln. Modifizierbar ist eine Dokumentation, wenn die Änderung einfach, vollständig und konsistent durchführbar ist.

```
eval> (define Modifizierbar?
        (lambda (Änderung)
          (and (Einfach-durchführbar?
                 Änderung)
               (Vollständig-durchführbar?
                 Änderung)
               (Konsistent-durchführbar?
```

```
                    Änderung))))
```

Die einfache Durchführbarkeit der Änderung setzt eine dem Benutzer vertraute Gliederung der Dokumentation voraus. Für die Quellcodetexte ist z. B. die Reihenfolge der einzelnen Konstrukte festzulegen. Analysieren wir eine Konstrukte-Hierarchie (↪ Abbildung 2.1 S. 149), dann folgen wir den Abstraktionsebenen; entweder von oben beginnend nach unten (*top-down*-Ansatz) oder umgekehrt, von unten beginnend nach oben (*bottom-up*-Ansatz). Dabei ist eine feste Reihenfolge innerhalb unseres Verbundes von Konstruktor, Selektor, Prädikät und Mutator einzuhalten.

Zusätzlich sind Verweise auf abhängige und redundante Text(teil)e erforderlich, damit die Änderung vollständig und konsistent durchführbar ist. Für die gesamte Dokumentation wird keine Redundanzfreiheit angestrebt. Sie widerspräche der Adressaten- und Zweck-Ausrichtung der Dokumente. Das schließt nicht aus, dass einzelne Dokumente (z. B. die Quick-Referenzkarte) möglichst redundanzfrei sind.

```
eval> (define Einfach-durchführbar?
        (lambda (Änderung)
          (and (Vertraute-Dokumentaions-Gliederung?
                 *BENUTZER*)
               (Verstehbar? Änderung)
               (Explizites-Verweissystem?
                 *DOKUMENTATION*))))
```

Die Eigenschaft *Zurückführbarkeit* bezieht sich auf die Konstruktions-Prozessdokumentation. Wir wollen zu einer Aussage in einem Dokument die zugehörigen Aussagen in der Kette der vorher gehenden Dokumente auffinden. Die *Zurückführbarkeit* verlangt, dass wir z. B. aus einem Konstrukt im Quellcodetext die betreffende Anforderung im Pflichtenheft identifizieren können.

Die einzelnen Dokumente dürfen kein Konglomerat, d. h. kein buntzusammengewürfeltes Papierwerk, bilden. Erst eine gegenseitige Abstimmung („Arbeitsteilung“) der einzelnen Dokumente macht eine Dokumentation handhabbar. Die Eigenschaft *Handhabbarkeit* betrifft den Umfang jedes einzelnen Dokumentes. Ein adressaten- und zweckgerechter Umfang ist nur erreichbar, wenn wir problemlos auf Aussagen anderer Dokumente verweisen können.

Die Vor- und Rückwärtsverkettung verbessert die *Modifizierbarkeit*, *Zurückführbarkeit* und *Handhabbarkeit* der Dokumentation. Voraussetzung ist dazu, dass jedes Dokument und jede Aussage in jedem Dokument mit einem eindeutigen Bezeichner (*Identifier*) versehen ist. Wir notieren daher keine Überschrift für ein Kapitel, einen Abschnitt oder Unterabschnitt, keine Unterschrift für eine Abbildung, Zeichnung oder Tabelle und keine wesentliche Aussage im Text ohne (alpha-)numerische Kennziffer.

Identifier: < *buchstabe* >< *nummern* >	
< *buchstabe* >	Art der Aussage
`A`	≡ Anforderung
`E`	≡ Entwurf
`I`	≡ Implementation
`T`	≡ Test
< *nummern* >	Gegliedert mittels Punkt
	z. B. : `1.8.3`

Tabelle 3.6: Vorschlag: Identifier in der Dokumentation

In welcher Form die Kennziffern als Basis dieser Vor- und Rückwärtsverkettung zu bilden sind, ist in den jeweiligen Dokumentationsrichtlinien festzuschreiben. In diesem Buch benutzen wir Zahlen mit einem vorangestellten Buchstaben (↪ Tabelle 3.6 S. 396). Außerdem haben die Dokumentationsrichtlinien vorzuschreiben, welche Kenndaten ein selbstständiges Dokument aufzuweisen hat (*Kopfdaten*- und *Fußdaten*-Vorschrift). Zumindest erforderlich sind die Angaben:

1. Dokumentnummer
2. Kurzbezeichnung
3. kurze Zweckangabe (Zusammenfassung) und
4. Erstellungs- und Modifikationsdaten.

Die folgende Gegenüberstellung einer Beschreibung weniger Anforderungen zeigt Möglichkeiten einer Identifizierung entsprechend Tabelle 3.6 S. 396.

Beispiel: Plausibilitätsprüfung

Fall 1: Verbaler Text (So nicht!) Das Programm soll drei Datenelemente auf Plausibilität überprüfen. Ist das erste Datenelement falsch, wird eine Fehlernachricht ausgegeben. Ist das zweite Datenelement nicht richtig wird ebenfalls eine Fehlermeldung ausgegeben. Ist das dritte Datenelement falsch, soll das Programm den Wert `Unbekannt!` annehmen.

Fall 2: Verbaler Text mit Bezeichnern

`A1`: Das Programm `Plausibel?` überprüft die drei Datenelemente: `Element-1`, `Element-2` und `Element-3`.

ET-Plausibel?		R1	R2	R3
B1	`Element-1` falsch?	J	–	–
B2	`Element-2` falsch?	–	J	–
B3	`Element-3` falsch?	–	–	J
A1	Fehlernachricht `F01`	X		
A2	Fehlernachricht `F02`		X	
A3	`Plausibel?` nimmt Wert `W01 ="Unbekannt!"` an.			X

Legende: (↪ Abschnitt 13 S. 80)
Mehrtreffer-Entscheidungstabelle mit der Abarbeitungsfolge: `R1`, `R2`, `R3`
- J ≡ Ja, Bedingung trifft zu.
- – ≡ Irrelevant, Anwort hat keinen Einfluss auf die Wahl der Regel.
- X ≡ Aktion ist zu vollziehen.

Tabelle 3.7: Beispiel: Mehrtreffer-ET `Plausibel?`

`A1.1`: Ist `Element-1` falsch, gibt `Plausibel?` die Fehlernachricht `F01` aus.

`A1.2`: Ist `Element-2` falsch, gibt `Plausibel?` die Fehlernachricht `F02` aus.

`A1.3`: Ist `Element-3` falsch, nimmt `Plausibel?` den Wert `W01 = "Unbekannt!"` an.

Fall 3: Formalisierter Text mit Bezeichern (↪ Tabelle 3.7 S. 397).

In den Fällen 2 und 3 ist jede einzelne Anforderung direkt angebbar. Jedes nachfolgende Dokument, z. B. der Quellcodetext, kann sich auf diese Identifizerung stützen. Bei einer Modifikation sind die Änderungsbereiche problemlos adressierbar, z. B. :

Fall 2: `A1.3: Wert W01 = "Unbekannt!"`

Fall 3: `ET-Plausibel? A3: Wert W01 = "Unbekannt!"`

3.1.4 Zusammenfassung: Namen, Kommentare und Verweise

Die Konstruktions- und Konstruktions-Prozessdokumente sind wesentliche Bestandteile der Software. Jede Aussage in jedem Dokument hat sich nach den Adressaten und dem Zweck zu richten.

Das Baukasten-Konstruktionsprinzip bedingt, dass Formulierungen mehrfach verwendbar sind. Daher sind z. B. Anforderungen im Präsens zu formulieren.

Der Name eines Konstruktes (z. B. einer Funktion, eines Makros oder einer Variablen) ist abhängig vom gewählten Paradigma. Einerseits verweist er auf das repräsentierte „reale“ Objekt und andererseits transportiert er konstruktionsspezifische Informationen. Letztere lassen sich gut mit einer Affix-Konvention verschlüsseln. Bei der Benennung einer Funktion ist stets die Informationsaufteilung mit anderen Namen (`lambda`-Variablen) und dem Eintrag im (Daten-)Lexikon zu bedenken.

Kommentarzeilen im Quellcodetext dienen primär zur *Warum*-Erläuterung. Mit einer verbindlichen Regelung ist festzulegen, worauf sie sich beziehen und welchen Textbereich ihre Erklärung umfasst.

Vor- und Rückwärtsverweise verbessern die Dokumentations-Eigenschaften *Modifizierbarkeit*, *Zurückführbarkeit* und *Handhabbarkeit*. Dazu ist jedes Dokument und jede Aussage in einem Dokument mit eindeutigen Bezeichnern zu versehen.

Charakteristisches Beispiel für Abschnitt 3.1

Hinweis: *PLT-Scheme*.
Zur Nutzung der Eigenschaftsliste (P-Liste ↪ Abschnitt 2.2.3 S. 191) laden wir vorab `compat.ss` — bei üblicher Installation also die Datei:
`C:\Programme\plt\collects\mzlib\compat.ss`
Um `set-car!`- und `set-cdr!`-Mutatoren (*Mutable Pairs*) nutzen zu können, laden wir dann noch die folgenden Bibliotheken aus Scheme R6RS.

```
(require rnrs/base-6)
(require rnrs/lists-6)
(require rnrs/mutable-pairs-6)
```

Programm `Diagnose`

`A1`: `Diagnose`, ein einfacher Regelinterpreter, analysiert Aussagen, die aus Bedingungen und Aktionen bestehen.

- `A1.1`: Aussagen sind aus Regeln abgeleitet oder werden
- `A1.2`: beim Benutzer nachgefragt.

`A2`: Eine Regel hat

- `A2.1`: eine oder mehrere Bedingungen und
- `A2.2`: eine Aktion.

`A3`: Eine Aussage ist eine Liste, bestehend aus dem Objekt, seiner Eigenschaft und deren Wert.

A4: `Diagnose` ermittelt für ein vorgegebenes Objekt und eine vorgegebene Eigenschaft den zugehörigen Wert und

A5: begründet diesen, durch Auflistung der Bedingungen von zutreffenden Regeln.

E1: Abgebildete Daten(strukturen):

E1.1: $<$ *aussage* $>\equiv$;vgl.A3
($<$ *objekt* $><$ *eigenschaft* $><$ *wert* $>$)

E1.2: $<$ *aussage* $>\equiv<$ *bedingung* $>|<$ *aktion* $>$;vgl. A1

E1.3: $<$ *regel* $>\equiv$ (($<$ *bedingung*$_1$ $>$;vgl. A2
$\ldots <$ *bedingung*$_n$ $>$) $<$ *aktion* $>$)

E1.4: $<$ *regeln* $>\equiv$ ($<$ *regel*$_1$ $> \ldots <$ *regel*$_n$ $>$)

E1.5: Regeln sind einer Eigenschaft zugeordnet:
`(putprop` $<$ *eigenschaft* $>$ $<$ *regeln* $>$ `'Regeln)`

E1.6: Der Wert kommt vom Benutzer, wenn: ;vgl.A1.2
`(putprop` $<$ *eigenschft* $>$ `'nachfragen #t)`

E1.7: Die Startaussage und die von `Diagnose` ermittelten Begründungen sind Listenelemente der Variablen `*AUSSAGEN*`

T1: Testdaten (↪ Abschnitt 3.1.4 S. 402)

T2: Testapplikation (↪ Abschnitt 3.1.4 S. 403)

```
eval> (define Diagnose
        (lambda (Objekt Eigenschaft)
      (letrec (

;;;;Primitive Konstruktoren und Selektoren ;vgl. E1
       (Make-Aussage
         (lambda (Objekt Eigenschaft Wert)
           (list Objekt Eigenschaft Wert)))
       (Get-Objekt
         (lambda (Aussage)
           (car Aussage)))
       (Get-Eigenschaft
         (lambda (Aussage)
           (list-ref Aussage 1)))
       (Get-Wert
         (lambda (Aussage)
           (list-ref Aussage 2)))
       (Get-Bedingungen
         (lambda (Regel)
           (car Regel)))
       (Get-erste-Bedingung
         (lambda (Bedingungen)
```

```
          (car Bedingungen)))
       (Get-Rest-Bedingungen
         (lambda (Bedingungen)
           (cdr Bedingungen)))
       (Get-aktion
         (lambda (Regel)
           (list-ref Regel 1)))
      (Get-Regeln
        (lambda (Eigenschaft) ;vgl. E1.5
          (getprop Eigenschaft 'Regeln)))
       (Get-erste-Regel
         (lambda (Regeln)
           (car Regeln)))
       (Get-Rest-Regeln
         (lambda (Regeln)
           (cdr Regeln)))
       ;;Regel ist relevant, wenn ihre
       ;; Aktion gleiches Objekt
       ;; hat wie die zu prüfende Aussage
      (Relevante-Regeln
        (lambda (Regeln Objekt Eigenschaft)
         (cond((null? Regeln) null)
              ((eq? #f Regeln) null)
              ((eq? Objekt
                    (Get-Objekt
                      (Get-aktion
                        (Get-erste-Regel
                          Regeln))))
                (cons (Get-erste-Regel Regeln)
                      (Relevante-Regeln
                       (Get-Rest-Regeln Regeln)
                         Objekt Eigenschaft)))
              (#t (Relevante-Regeln
                    (Get-Rest-Regeln Regeln)
                      Objekt Eigenschaft)))))

;;;;Konstrukte zum Fortschreiben
;;;; von *AUSSAGEN* ;vgl. E1.7
      (Ergaenzung-*AUSSAGEN*!
        (lambda (neue_Aussage)
         (set! *AUSSAGEN*
           (cons neue_Aussage *AUSSAGEN*))))

      (Modifikation-*AUSSAGEN*!
        (lambda (alte_Aussage neue_Aussage)
         (set-car! alte_Aussage neue_Aussage)))

       ;;Effizientes Suchen durch
       ;; zweistufiges Verfahren
      (Get-Wert-in-*AUSSAGEN*
        (lambda (Objekt Eigenschaft)
         (do ((fakt (assoc Objekt *AUSSAGEN*)
                    (assoc Objekt (cdr fakt))))
             ((or (eq? fakt #f)
                  (eq? (Get-Eigenschaft fakt)
                       Eigenschaft))
```

```
              (if (eq? fakt #f)
                  #f
                 (Get-Wert fakt))))))

      (Aussage-in-*AUSSAGEN*!
       (lambda (aktion)
        (let ((bisheriger-Wert
               (Get-Wert-in-*AUSSAGEN*
                  (Get-Objekt aktion)
                  (Get-Eigenschaft aktion))))
          (cond((or (null? bisheriger-Wert)
                    (eq? #f bisheriger-Wert))
                (Ergaenzung-*AUSSAGEN*! aktion))
               (#t (Modifikation-*AUSSAGEN*!
                     bisheriger-Wert
                     aktion))))))

;;;;Wert beim Benutzer nachfragen ;vgl. E1.6
;;;; (benutzerschnittstelle)
      ;;Voraussetzung für eine
      ;; Benutzernachfrage
      (nicht-ableiten?
       (lambda (Eigenschaft)
        (getprop Eigenschaft 'nachfragen)))

      (benutzer-rueckfrage!
       (lambda (Objekt Eigenschaft)
         (display "Nenne ")
         (display Eigenschaft)
         (display " von ")
         (display  Objekt)
         (display "? ")
         (Ergaenzung-*AUSSAGEN*!
            (Make-Aussage
              Objekt
              Eigenschaft
             (read)))))

;;;;Regelauswahl, Bedingungen testen,
;;;; neues Ziel bilden
      (test   ;b_n ::= Bedingungen (plural)
       (lambda (b_n)
        (cond((null? b_n) #t)
             ((member (Get-erste-Bedingung b_n)
                *AUSSAGEN*)
              (test (Get-Rest-Bedingungen b_n)))
             ((Zielbildung
              (Get-Objekt
                (Get-erste-Bedingung b_n))
              (Get-Eigenschaft
                (Get-erste-Bedingung b_n)))
              ;;Zielbildung kann *AUSSAGEN*
              ;; modifizieren, daher
              ;; erneutes Nachschauen
              ;; in *AUSSAGEN*
              (cond((member
```

```
                    (Get-erste-Bedingung b_n)
                       *AUSSAGEN*)
                    (test
                      (Get-Rest-Bedingungen b_n)))
                      (#t #f)))
            (#t #f))))

      (pruefe-Regeln
       (lambda (Regeln)
        (cond((null? Regeln) null)
             ((test (Get-Bedingungen
                (Get-erste-Regel Regeln)))
              (Aussage-in-*AUSSAGEN*!
                 (Get-aktion
                   (Get-erste-Regel Regeln))))
             (#t (pruefe-Regeln
                   (Get-Rest-Regeln Regeln))))))

      (Zielbildung
       (lambda (akt_Objekt akt_Eigenschaft)
        (cond((Get-Wert-in-*AUSSAGEN*
                  akt_Objekt
                  akt_Eigenschaft))
             ((nicht-ableiten? akt_Eigenschaft)
              (benutzer-rueckfrage!
                akt_Objekt akt_Eigenschaft))
             ((pruefe-Regeln
                 (Relevante-Regeln
                    (Get-Regeln
                      akt_Eigenschaft)
                    akt_Objekt
                    akt_Eigenschaft)))
             (#t #f)))))
   ;;Starten der Regelabarbeitung
  (Zielbildung Objekt Eigenschaft)
  *AUSSAGEN*)))
```

Beispieldaten für das Programm `Diagnose`

```
eval> (define *AUSSAGEN*
        '((Benzinpumpe arbeitet korrekt)))
eval> (putprop
      'arbeitet
      'Regeln
      '((((Batterie Strom aussreichend)
          (Kabel Kontakte leitend))
         (Motor arbeitet gut))
        (((Benzinversorgung arbeitet nicht))
         (Motor arbeitet nicht))
        (((Lichtmaschine arbeitet nicht))
         (Motor arbeitet nicht))
        (((Tankfuellung Stand ausreichend)
          (Benzinpumpe arbeitet korrekt)
```

```
          (Benzinfilter Durchlaessigkeit keine))
         (Benzinversorgung arbeitet nicht))))
eval> (putprop 'Stand
                'nachfragen #t)
eval> (putprop 'Durchlaessigkeit
                'nachfragen #t)
```

Beispiel einer Applikation von `Diagnose`:

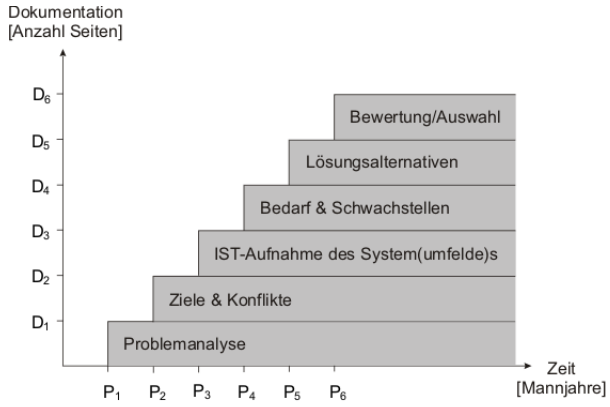
```
eval> (Diagnose 'Motor 'arbeitet) ==>
                                ;Benutzereingabe
Nenne Stand von Tankfuellung? ausreichend
                                ;Benutzereingabe
Nenne Durchlaessigkeit von Benzinfilter? keine
                                ;Ergebnis
{{Motor arbeitet nicht}
 {Benzinversorgung arbeitet nicht}
 {Benzinfilter Durchlaessigkeit keine}
 {Tankfuellung Stand ausreichend}
 {Benzinpumpe arbeitet korrekt}}
```

3.2 Spezifikation mit LISP

Spezifizieren heißt eine (zunächst noch unbekannte) Aufgabenstellung durch sprachliche Festlegung ihres Ergebnisses, bzw. ein (zunächst noch unbekanntes) Objekt durch sprachliche Festlegung seines Gebrauchs zu präzisieren (↪ z. B. [90]).

Publikationen zur Thematik *Software-Spezifikation* gehen von unterschiedlichen Begriffsdefinitionen aus. Zumeist basieren sie auf der Abgrenzung einzelner Phasen des Lebenszyklus (↪ Abbildung 1.13 S. 51) und der Beschreibung ihrer Dokumente. Zielvorstellung ist dabei die Spezifikation mit der Präzision der Mathematik und der Verständlichkeit der Umgangssprache.

Spezifizieren im Sinne eines Präzisierens umfasst alle Phasen. In den ersten Phasen bezieht sich die Spezifikation auf die Analyse und Synthese möglicher Konzeptionen. Ausgehend von einer Problemanalyse, werden mit der Aufnahme der Ziele und Konflikte, sowie der Ist-Situation des bisherigen System(umfelde)s, der Bedarf und die Schwachstellen präzisiert. Alternative Lösungansätze sind dann anhand der präzisierten Ziele und Konflikte zu bewerten (↪ Abbildung 3.1 S. 404). Anschließend folgt das Spezifizieren im engeren Sinne, d. h. das Entwerfen und Formulieren von Quellcodetexten. Konzentriert man sich auf die Phase, in der anwendungsspezifische Datentypen zu präzisieren sind, dann besteht die Spezifikation aus Sorten, Operationen und Gleichungen. Diese Spezifikation notiert die Datentypen im Sinne von Algebren (Näheres zur algebraischen Spezifikation ↪ z. B. [111]). Wir gehen von einer



Legende:
$D_i \equiv$ Dokument als Ergebnis der Phase P_i.

Abbildung 3.1: Spezifizieren in den ersten Phasen

gewählten Lösungsalternative aus (↪ Abbildung 3.1 S. 404, nach Phase P_6) und spezifizieren schrittweise die Aufgabenstellung so weit, bis diese (ohne Missverständnisse) direkt konstruierbar (definierbar) ist. Diese Spezifikation präzisiert: Was leistet die Konstruktion im Einzelnen? Die Hauptaufgabe ist somit das Erarbeiten detaillierter Anforderungen (↪ Tabelle 3.8 S. 405).

Die Anforderungen können sich auf numerische Berechnungen (z. B. Lösungen von Differentialgleichungen), geometrische Berechnungen (z. B. 3-dimensionale Modelle), Datenaggregationen (z. B. Besoldungsbescheide), Transaktionsprozesse (z. B. Platzreservierungssysteme), Wissensakquisition (z. B. Diagnosesysteme), Kommunikationsprozesse (z. B. Elektronische Post) und vieles mehr beziehen.

Diese Vielfalt der Konstruktionsaufgaben macht unterschiedliche Arbeitstechniken notwendig. Arbeitstechniken umfassen einerseits Prinzipien (Handlungsgrundsätze) und Methoden (erfolgversprechende, begründbare Vorgehensweisen) und andererseits Instrumente (*Werkzeuge*). Erste sind nicht, zweite sind (teilweise) selbst Software.

Empfehlungen zur Spezifikation haben zumindest die Aufgabenart und die Aufgabengröße zu berücksichtigen. Es ist z. B. nicht unerheblich, ob ein administratives System oder ein System zur Steuerung eines technischen Prozesses zu spezifizieren ist.

Im ersten Fall ist das Nutzen/Kostenverhältnis einzelner Systemleistungen disponibel bzw. unklar. Wir haben es mit Definitionsproblemen der Automationsaufgabe zu tun.

Im zweiten Fall mag die geforderte Reaktionsgeschwindigkeit das ei-

Anforderungskategorie	Kernfrage	Beispielformulierungen (↪ Programm `Diagnose` Abschnitt 3.1.4 S. 398)
1. funktionale Anforderungen	Was soll das System tun?	Aussagen sind aus Regeln abgeleitet oder werden beim Benutzer nachgefragt.
2. Anforderungen an des fertige Produkt	Was ist das System?	Das Dialogsystem `Diagnose` ist ein Produkt für den Einsatz bei mehr als 100 Kfz-Betrieben.
3. Anforderungen zur Systemumgebung	Was sind die Einsatzbedingungen?	`Diagnose` setzt das Betriebssystem *MS Windows Version 7* voraus.
4. Anforderungen an die Zielstruktur	Was ist der Bewertungshintergrund?	`Diagnose` verkürzt die Fehlersuchzeit zumindest bei angelernten Kräften.
5. Anforderungen zur Projektdurchführung	Was sind die Ressourcen für das Projektmanagement?	`Diagnose` Version 1.0 ist innerhalb von 3 Monaten mit 2 Mann zu entwickeln.

Tabelle 3.8: Kategorien von Anforderungen

gentliche Konstruktionsproblem sein. Das Ziel und die Arbeitstechnik seien klar; das Problem liegt in der Durchführung (Vollzugsproblem). Ausgehend von unserem (Vor)-Wissen über die erfolgversprechenden Arbeitstechnik und der Klarheit der Zielkriterien, sind verschiedene Problemfelder zu unterscheiden (↪ Tabelle 3.9 S. 406).

Unsere Konstruktionsaufgabe sei ein Vollzugsproblem (↪ Feld I.1 in Tabelle 3.9 S. 406). Die erfolgversprechende Arbeitstechniken zur Spezifikation beziehen sich damit auf:

1. eine schrittweise Verfeinerung („vertikale Ebene“) und
2. eine modulare Strukturierung („horizontale Ebene“)

von Anforderungen. Für beide Fälle bedürfen wir einer Rechnerunterstützung.

Die Konstruktionsgröße bestimmt welche (rechnergestützten) Arbeitstechniken anzuwenden sind. Bei einer kleinen Konstruktionsaufgabe ist die Spezifikation der einzelnen Verarbeitungsprozesse und der Datenrepräsentation zu meistern. Bei einer sehr großen Aufgabe sind zusätzlich die Fortschritte der Systemsoftware und Hardware während der benötigten Planungs- und Realisierungszeit einzukalkulieren. Außerdem ändern sich Anforderungen in dieser relativ langen Zeit. Zu spezifizieren ist daher eine Weiterentwicklung, d. h. ein Evolutionskonzept. Tabelle 3.10 S. 406 skizziert benötigte Arbeitstechniken in Abhängigkeit zur Konstruktionsgröße.

In der Tabelle 3.10 S. 406 ist der Umfang des geschätzten Quellcodetextes nur ein grober Maßstab. Die Angabe des Aufwandes in *Man-*

Zu Projektbeginn *Klarheit* über:				
die Arbeitstechniken			die Ziele	
			groß I	gering II
1	erfolgversprechende *Prinzipien*, *Methoden* und *Instrumente* zur Problemlösung	bekannt	*Vollzugs*-probleme	*Definitions*-probleme der Automationsaufgabe
2		unklar	*Zielerreichungs*-probleme	*Steuerungs*-probleme eines kontinuierlichen Herantastens

Tabelle 3.9: Problemarten beim Spezifizieren

Konstruktions-Kategorie		Konstruktionsgröße [LOC]	Aufwand [MJ]	Erforderliche Arbeitstechniken (*Prinzipien*, *Methoden*, *Instrumente*) zur:
1	Kleine Konstruktion	< 1.000	< 0.5	⋆ funktionalen Strukturierung ⋆ Datenrepäsentation
2	Mittlere Konstruktion	< 10.000	< 4	⋆ Projektplanung ⋆ Projektüberwachung für den gesamten Lebenszyklus ⋆ Anforderungsanalyse (*Requirements Engineering*) ⋆ plus (1)
3	Große Konstruktion	< 100.000	< 25	⋆ Durchführbarkeitsstudie ⋆ Definition von Datennetzen und Datenbankmanagement ⋆ Konfigurationsmanagement ⋆ plus (2)
4	Sehr große Konstruktion (noch größer „zerfallen“ in eigenständige Teile)	> 100.000	> 25	⋆ Softwareevolution ⋆ Hardwareevolution ⋆ Dynamik der Anforderungen ⋆ plus (3)

Legende:
LOC ≡ Umfang der Quellcodetexte (*Lines of Code*)
MJ ≡ *Manpower*-Jahre

Tabelle 3.10: Konstruktions-Kategorien und Arbeitstechniken

power-Jahren (kurz: *MJ*) ist umstritten und berechtigt kritisierbar ("*The Mythical Man-Month*"; ↪ [30]). Hier dienen die LOC- und MJ-Werte nur zur ganz groben Unterscheidung, ob ein kleines oder großes Team die Aufgabe bewältigen kann. Bei einer größeren Anzahl von Programmierern ist eine gößere Regelungsdichte erforderlich. Die Dokumentationsrichtlinien, die Arbeitsteilung und die Vollzugskontrollen sind entsprechend detailliert zu regeln. Es bedarf einer umfassenden Planung und Überwachung des gesamten Lebenszyklus (↪ Abbildung 1.13 S. 51).

Im folgenden behandeln wir funktionale Anforderungen bei einem Vollzugsproblem und unterstellen eine kleine bis mittlere Konstruktionsgröße, so dass wir z. B. auf Angaben zum Konfigurationsmanagement verzichten können. Vorab wird die Frage diskutiert: Welche Voraussetzungen sind zu erfüllen, damit eine Aussage eine gestaltungsrelevante Anforderung ist? Skizziert sind daher Anforderungen an Anforderungen (↪ Abschnitt 3.2.1 S. 407). Ihre Strukturierung geschieht durch die Modularisierung mit Hilfe der Definition von Import-Export-Schnittstellen (↪ Abschnitt 3.2.2 S. 411).

3.2.1 Anforderungen an Anforderungen

Funktionale Anforderungen bedingen, dass es sich um Aussagen über ein *System* handelt. Auch ohne Definition vermittelt der Begriff *System* die Intention Mannigfaltiges als Einheit zu betrachten. Ein System ist daher abgrenzbar von seiner Umwelt und hat Bestandteile, zwischen denen Wechselwirkungen (Interaktionen) stattfinden. Diese Merkmale sind als Prädikat, wie folgt, formulierbar.

```
eval> (define System? (lambda (X)
        (and
         (|Abgrenzbare Einheit gegenüber der Umwelt?|
           X)
         (|Gliederbar in elementare Bestandteile?|
           X)
         (|Interaktionen zwischen Bestandteilen?|
           X))))
```

Eine funktionale Anforderung beschreibt eine vom System zu erfüllende Leistung, bzw. eine quantiative oder qualitative Eigenschaft. Solche Aussagen erfüllen daher folgendes Prädikat:

```
eval> (define Anforderung?
        (lambda (Aussage)
          (and
           (|Spezifiziert System?| Aussage)
            (or
             (|Spezifiziert zu erfüllende Leistung?|
                Auassage)
             (|Spezifiziert quantitative Eigenschaft?|
```

```
    Aussage)
  (|Spezifiziert qualitative Eigenschaft?|
    Aussage)))))
```

Wir diskutieren vier exemplarische Aussagen, von denen wir unterstellen, dass sie das Prädikat `Anforderung?` erfüllen mögen. Sie beziehen sich auf den einfachen Regelinterpreter `Diagnose` (↪ Abschnitt 3.1.4 S. 398).

`A1`: `Diagnose` fragt Fakten beim Benutzer nach.

`A2`: `Diagnose` ist benutzerfreundlich.

`A3`: `Diagnose` macht den Benutzer glücklich.

`A4`: `Diagnose` ist `Diagnose` (in Anlehnung an: *Volvo ist Volvo*[6]).

Die Aussagen `A1` bis `A4` unterscheiden sich erheblich in ihrer Realisierbarkeit. Die Aussage `A1` ist eine zu erfüllende Leistung, die bei verfeinerter Spezifikation zur Definition der Konstrukte `Benutzer-Rueckfrage!` und `Nicht-ableiten?` führt (↪ Abschnitt 3.1.4 S. 398).

Die Eigenschaft „Benutzerfreundlichkeit“ bezeichnet eine Qualität. Die (objektive) Definition von Qualität wird in der Philosophie oft als unmöglich angesehen (↪ z. B. [142]). Die Spezifikation einer Qualität ist daher stark vom jeweiligen Konstrukteur abhängig. Für uns soll „Benutzerfreundlichkeit“ einerseits niedrige subjektive Bedienungskomplexität (für den ungeübten Benutzer) und andererseits große Nutzungsflexibilität (für den geübten Benutzer) umfassen. Aspekte sind dabei die Darstellungsergonomie (Kognitionsproblem), Ablaufergonomie (Konzentrationsproblem) und die Funktionsergonomie (Produktivitätsproblem).

Die Aussage `A2` ist daher im Vergleich zu `A1` stärker interpretationsbedürftig. Die qualitative Eigenschaft „Benutzerfreundlichkeit“ ist durch mehrere „abgeleitete“ Anforderungen zu approximieren. So ist „Benutzerfreundlichkeit“ z. B. gliederbar in `Erlernbarkeit` und *Handhabbarkeit/Effektivität*. Letztere ist approximierbar durch *Einfachheit der Nutzung*, *Durchschaubarkeit*, *Komfort*, *Flexibilität* und *Bedienerbezogene Robustheit* (↪ [12]). *Komfort* wäre z. B. realisierbar durch eine Ein-/-Ausgabe-Fenstertechnik. Schrittweise präzisieren wir `A2` durch mehrere qualitative Eigenschaften für die Ersatzleistungen angebbar sind. Die Anforderung `A2` ist (näherungsweise) ersetzbar durch eine Menge von Anforderungen, die zu erfüllende Leistungen spezifizieren. Eine von diesen wäre z. B. :

`A2.i`: `Diagnose` zeigt die jeweils möglichen Kommandos in einem eigenen Bildschirmfenster.

[6] *Volvo* (lat.: *ich rolle*) ist ein ursprünglich schwedischer Fahrzeugkonzern. Der Konzern wurde 1927 als Pkw-Hersteller gegründet. Sein plakatierter Werbespruch stammt von ≈ 1990.

Die Aussage A3 ist sicherlich erstrebenswert; jedoch nicht wie A2 durch realisierbare funktionale Aussagen approximierbar. Die Aussage A4 mag werbewirksam sein, ist aber inhaltsleer. Beide Aussagen haben keinen Einfluss auf die Konstruktion von Diagnose.

Um utopische oder inhaltsleere Aussagen auszuschalten, setzen wir für eine Anforderung voraus, dass sie die Konstruktion tatsächlich beeinflusst (gestaltet). Solche gestaltungsrelevanten Aussagen bezeichnen wir im Folgenden als *Vorgaben*. Für eine Vorgabe ist (zumindest iterativ) ein Realisierungsplan entwickelbar, der ordnungsgemäß vollzogen, zu einem akzeptablen Ergebnis führt. Ein Ergebnis ist akzeptabel, wenn die *SOLL-IST*-Abweichung eine Toleranzgrenze nicht überschreitet, also kleiner einer vorgegebenen duldbaren Abweichung ist. Wir definieren ein entsprechendes Prädikat, wie folgt:

```
eval> (define Realisierbar?
        (lambda (Anforderung |Duldbare Abweichung|)
          (<= (|Bestimme SOLL-IST-Abweichung|
        (|Feststellen IST-Verhalten bezogen auf SOLL|
                  (Planvollzug
                    (Planung
                     Anforderung
                      |Duldbare Abweichung|)))))
                |Duldbare Abweichung|)))
```

Entsprechend diesem Prädikat kann die |Duldbare Abweichung| oder die Anforderung selbst modifiziert werden, um Realisierbarkeit zu erreichen. Die Aussage A2 („Benutzerfreundlichkeit“) zeigt, dass auch die Modifikation einer Anforderung in gewissen Interpretationsgrenzen in Betracht kommen kann. Statt einen Teil von A2 mit A2.i (überlappende Bildschirmfenster) zu approximieren, ist ein Teil von A2 als „Kommandos über Funktionstasten“ (A2.j) interpretierbar. In der Regel ist jedoch die Toleranzschwelle das Mittel, um eine Realisierbarkeit noch zu erreichen.

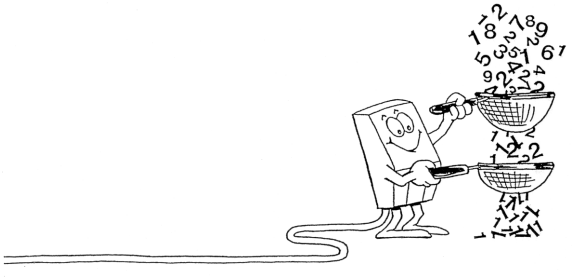
Für eine Vorgabe können wir daher folgendes Prädikat definieren:

```
eval> (define Vorgabe?
(lambda (Anforderung |Duldbare Abweichung|)
       (letrec
 ((Anforderung? (lambda (Aussage)
   (and (|Spezifiziert System?| Aussage)
             (or
              (|Spezifiziert zu erfüllende Leistung?|
                 Aussage)
              (|Spezifiziert quantitative Eigenschaft?|
                 Aussage)
              (|Spezifiziert qualitative Eigenschaft?|
                 Aussage)))))
            (Realisierbar?
              (lambda
```

```
              (Anforderung |Duldbare Abweichung|)
               (cond(
       (<= (|Bestimmung SOLL-IST-Abweichung|
        (|Feststellen IST-Verhalten bezogen auf SOLL|
                  (Planvollzug
                    (Planung
                      Anforderung
                      |Duldbare Abweichung|))))
            |Duldbare Abweichung|) #t)
       ((Modifizierbar? Anforderung)
                 (Realisierbar?
                       (Modifikation Anforderung)
                        |Duldbare Abweichung|))
                  ((Modifizierbar?
                     |Duldbare Abweichung|)
                      (Realisierbar?
                        Anforderung
                        (Modifikation
                          |Duldbare Abweichung|)))
                  (#t #f)))))
(and (Anforderung? Anforderung)
              (Realisierbar?
                Anforderung
                |Duldbare Abweichung|))))))
```

Die einzelne Anforderung mag das Prädikat `Vorgabe?` erfüllen; im Zusammenhang mit anderen jedoch nicht. Naturgemäß gibt es konkurrierende und sich widersprechende Vorgaben. Vorrang- und Kompromißentscheidungen sind daher im Rahmen der Spezifikation zu treffen. Eine pragmatische Vorgehensweise beachtet bei den einzelnen Spezifikationsschritten vorrangig kritische Vorgaben und gewährleistet die anderen quasi nachträglich durch gezielte Modifikationen. Die kritische Vorgaben intuitiv zu erkennen, ist die „Kunst“ des erfahrenen Konstrukteurs.

Ziel der Spezifikation ist es, die Vorgaben so auszuarbeiten, dass sie durch passende Verknüpfungen möglicher Konstrukte abbildbar werden. Dazu bedarf es ihrer Strukturierung in Teilbereiche, die dann weitgehend unabhängig von einander mehr und mehr präzisiert werden. Die Konstruktion wird als „Geflecht“ von Konstrukten (*Moduln*), die über Schnittstellen miteinander verknüpft sind, spezifiziert. Wir verwenden hier den üblichen Begriff *Modul* im Sinne eines Synonyms für Konstrukt bzw. Teilkonstruktion. Die „Dopppelgesichtigkeit“ jeder Schnittstelle, die auch der englische Begriff *interface* betont, bezieht sich auf die Menge aller Annahmen, die jeweils der eine Modul vom anderen unterstellt und umgekehrt. Innerhalb der Schnittstelle werden daher die relevanten Eigenschaften für ihre Kommunikation präzisiert.



3.2.2 *Import/Export*-Beschreibung

Die Wechselwirkungen zwischen den einzelnen Moduln sind mit Hilfe einer *Benutzt*-Relation und einer *Enthält*-Relation beschreibbar. Die *Enthält*-Relation verweist auf die Möglichkeit, dass ein Modul selbst wieder in Module zerlegbar sein kann. Die *Benutzt*-Relation verweist auf die Modul-Schnittstelle.

Die Modul-Schnittstelle ist gliederbar in einen Import- und einen Export-Teil. Die Import-Schnittstelle beschreibt welche Konstrukte der Modul aus anderen Moduln benutzen kann. Die Export-Schnittstelle nennt die Konstrukte, die der Modul für andere Moduln bereitstellt. Wir sprechen daher von einem Import, wenn ein Modul m_{import} eine Leistung eines Moduls m_{export} verwendet. Der Modul m_{import} importiert (benutzt) ein Konstrukt, das m_{export} exportiert.

Bei der Diskussion des Verbundes von Konstruktor, Selektor, Prädikat und Mutator haben wir eine Hierarchie unterstellt (↪ Abbildung 2.1 S. 149). Sie ist im Sinne der *Benutzt*-Relation zyklenfrei, d. h. aus „m_{import} *benutzt* m_{export}“ folgt, dass m_{export} nicht m_{import} benutzt und dass kein in der Hierarchie unter m_{export} befindlicher Modul den Modul m_{import} benutzt. Außerdem gibt es einen hervorgehobenen Modul (Hauptmodul) mit einem besonderen Konstrukt, das als erstes appliziert wird. Der gesamte übrige Berechnungsvorgang (dynamische Ablauf) erfolgt über *Benutzt*-Relation.

Konstrukte des *PLT-Scheme* Modul-Konzeptes sind in Abschnitt 2.7 S. 319 beschrieben. Das dort skizzierte *Import/Export*-Konzept lässt sich zu einem *Vertragskonzept* (*Design by Contract* ↪ z. B. [129]) erweitern. Damit kommt man letztlich zur Programmierentwicklung auf der Basis von mathematisch formulierten Spezifikationen (*algebraic specifications*[7]).

Das `provide/contract`-Konstrukt ermöglicht die Notation von umfassenden Vertragsbedingungen. Das folgende Beispiel zeigt, dass das `Guthaben` im Modul `Foo` gemäß Vertrag stets eine positive Zahl sein muss. Im Modul `Bar` wird das `Guthaben` für die Berechnung des Umsat-

[7] Sie hatten in den 70iger Jahre ihren „*Hype*“.

zes genutzt und dabei werden die Vertragsbedingungen bezüglich `Foo` überprüft.

```
eval> (module Foo scheme
        (provide/contract
         (Guthaben (and/c number? positive?)))
        (define Guthaben "Alles klar?"))
eval> (module Bar scheme
        (require 'Foo)
        (provide/contract
          (Umsatz positive?))
        (define Umsatz (* 2 Guthaben)))
eval> (require 'Bar) ==> ERROR ...
      ;'Foo broke the contract
      ; (and/c number? positive?) on Guthaben;
      ; expected <(and/c number? positive?)>,
      ; given: "Alles klar?"
```

In der Praxis geschieht das Spezifizieren der Moduln zunächst „skelettartig“ mit lose verknüpften, verbalen Beschreibungen und Absichtserklärungen. Darauf folgt eine Phase der Konsolidierung, in der Schnittstellen und Implementationansätze verbindlich vereinbart werden. Anschließend sind diese Beschreibungsrahmen („Skelette“) auszufüllen. Das Ausfüllen kann mehr oder weniger lokal sequentiell erfolgen. Erforderlich ist allerdings die Bezugnahme auf andere, bereits teilweise erstellte „Skelette“.

In Anlehnung an das *Import/Export*-Konzept von *PLT Scheme* wird dazu folgender Beschreibungsrahmen[8], der die Aufgabe einer Checkliste hat, empfohlen:

(`module` $<name><initial-import>$
$\{<interface>\}$
(`define effects` $<text>$)
$\{<modul-functions>\}$)

mit:

$<name>$	$\equiv$ Mnemotechnisch prägnanntes Symbol, das sich von allen anderen unterscheidet.
$<initial-import>$	$\equiv$ Import der Grundlage, hier oft `scheme`.
$<interface>$	$\equiv$ Schnittstelle zu Moduln auf höherer und tieferer Hierarchie-Ebene mittels `provide`- und `require`-Konstrukten.
$<text>$	$\equiv$ Freier Text zur Kurzbeschreibung des Moduls.
$<modul-functions>$	$\equiv$ Beschreibung einzelner Funktionen

Mit der *Import/Export*-Unterscheidung bei der Schnittstelle entsteht daraus folgendes praxisorientierte „Skelett“. Dabei verdeutlichen die fett-

[8] *BFN*-Notation $\hookrightarrow$ Abschnitt 1.2.2 S. 65.

gedruckten Begriffe die Angaben, auf die man sich im jeweiligen Spezifikationsschritt konzentriert.

```
(module < name > scheme
  (provide ... < function_i > ...)
  (require ... < modul_j > ...)
  (define effects '|Text zur allgemeinen Beschreibung|)
  ({< modul − functions >}) )
```

Ausgangspunkt ist ein eindeutiger, kennzeichnender Modulname (↪ Abschnitt 3.1.1 S. 377). Mit der Eintragung unter `effects` beschreiben wir alle zu erfüllenden Leistungen und Eigenschaften des gesamten Moduls in Form eines kurzen Textes.

Die `provide`- und `require`-Angaben definieren die Modul-Hierarchie. In der Praxis können wir selten diese vom Hauptmodul ausgehende Hierarchie konsistent aufbauen. Neben den *top-down*-Schritten entsteht unser Skelett *bottom-up*, *middle-out*, *inside-out*, *outside-in* und/-oder *hardest-first*. Die `provide`- und `require`-Schnittstellen beschreiben wir daher eher Zug um Zug mit laufenden Korrekturen.

Während im Abschnitt `effects` der Modul insgesamt beschrieben ist, sind die einzelen zu erfüllenden Leistungen, hier als $< modul - function >$ bezeichnet, zunächst durch einen kennzeichnenden Namen anzugeben. Im nächsten Verfeinerungsschritt sind die Funktionen des Moduls weiter zu präzisieren und zwar durch die Beschreibung ihrer Signatur (Ein- und Ausgabe) in Form der `lambda`-Variablen und dem Funktionswert (Rückgabewerte). Für eine Beschreibung von Einschränkungen (*constraints*) und und von Nebeneffekten (*side-effects*) verwenden wir entsprechende Symbole. So können wir verdeutlichen, welche Veränderungen bleibende Wirkung haben, also länger leben als die Aktivierung des Moduls (Dauerhaftigkeit, Persistenz). Damit ergibt sich folgendes Beschreibungsraster:

```
(define < modul − function >
  (lambda ( {< variable >}
    (define contraints    '(< text >))
    (define side-effects '(< text >))
      ...  < value > )))
```

Dieses Spezifizieren ist ein Wechselspiel von Problemdurchdringung (Analyse), Erarbeitung einer Modulhierarchie (Synthese) und Überprüfung von Modulbeschreibungen (Revision). In diesem Wechselspiel entstehen neue, zum Teil bessere Ideen, die häufig eine Umstrukturierung bedingen. Der Beschreibungsrahmen erleichtert das Modifizieren. Mit Hilfe des Texteditors sind die einzelnen Slots des Rahmens leicht „um-

hängbar“.

Beispiel: `Fürsorge` **— Automation im Sozialamt einer Kommune** Das System `Fürsorge` rationalisiert das Zahlbarmachen von Sozialhilfe. Es unterstützt die Antragstellung, die Aktenverwaltung und die Aktualisierung von Berechnungsgrundlagen und Regelsätzen.

Die Anforderungen seien so weit erarbeitet, dass der Entwurf einer Modul-Hierarchie zu spezifizieren ist. Wir skizzieren diese grob, wie folgt:

```
;;;Nur als Platzhalter
eval> (module Regelsätze scheme)
eval> (module Berechnungsgrundlage scheme)
eval> (module Antragsteller scheme)
;
eval> (module Zugangsprüfung scheme
  (provide Befugnis)
  (define effects
    '(Sozialamtsleiter, Abteilungsleiter,
    Sachbearbeiter und Mitarbeiter
    haben unterschiedliche  Befugnisse.
    Zugangsprüfung verwaltet
    entsprechende Passwörter))
  (define |Fortschreibung Passwort|
    (lambda (...) ...))
  (define Befugnis
    (lambda (...) ...)))
;
eval> (module Aktenverwaltung scheme
  (provide Vorgang)
  (require 'Antragsteller)
  (define effects
    '(Aktenverwaltung  speichert für
    jeden Antragsteller alle ihn
    betreffenden Vorgänge unabhängig
    vom Sachbearbeiter in einer Akte.
    Für festgestellte Ansprüche
    generiert Aktenverwaltung die
    Auszahlungsanordnung. Das
    Aktenzeichen, Name des
    Antragstellers oder der
    eines betroffenen Familienmitgliedes
    sind Selektionsmerkmale))
  (define Vorgang
    (lambda (...) ...))
  (define Auszahlungsanordnung
    (lambda (...) ...)))
;
eval> (module Anspruchsanalyse scheme
```

```
  (provide Bescheid)
  (require 'Regelsätze 'Berechnungsgrundlage)
  (define effects
    '(Anspruchsanalyse ermittelt
     aus den geprüften  Antragsdaten
     Beträge und Zahlungstermine,
     falls  Ansprüche bestehen.))
  (define Bescheid
    (lambda (...) ...))
  (define Rückrechnung
    (lambda (...) ...)))
;
eval> (module Fürsorge scheme
  (provide |Formularbezogene Datenpflege|)
  (require 'Anspruchsanalyse
           'Aktenverwaltung
           'Zugangsprüfung)
  (define effects
    '(Fürsorge  prüft Anträge auf Zahlung von
    Sozialhilfe, erstellt  termingerecht
    Auszahlungsanordungen und führt die Akten.
    Fürsorge weist unberechtigte
    Terminalbenutzer ab.))
  (define |Formularbezogene Datenpflege|
     (lambda (...) ...))
  (define Rechtsbelehrung
     (lambda (...) ...)))

;;Top-level Nutzung
eval> (require 'Fürsorge)
eval> (|Formularbezogene Datenpflege| "Alles klar?") ==>
      "Alles klar?"
```

Im Rahmen der weiteren Verfeinerung könnten wir z. B. folgenden Modul `Anspruchsanalyse` erhalten. Er zeigt, dass zu diesem Zeitpunkt die einzelnen Modulfunktionen unterschiedlich fein spezifiziert sind.

```
eval> (module Regelsätze scheme
   (provide Wohngeld Sozialhilfe)
   (define effects '(Regelsätze ...))
   (define Wohngeld
     (lambda (...) ...))
   (define Sozialhilfe
     (lambda (...) ...)))

eval> (module Anspruchsanalyse scheme
  (provide Bescheid)
  (require (only-in 'Regelsätze Wohngeld Sozialhilfe)
           'Berechnungsgrundlage)
```

```
  (define effects
    '(Anspruchsanalyse ermittelt
     aus den geprüften  Antragsdaten
     Beträge und Zahlungstermine,
     falls  Ansprüche bestehen.))
;
  (define Bescheid
    (lambda (|Daten Formular 13|
             |Akte Antragsteller|)
      (define effects
        '(Berechnet Beträge/Termine und prüft
        die Akte im Hinblick auf Überzahlung.))
      (define constraints
        '(Die einzelnen Berechnungsschritte
        sind über die Dialogschnittstelle
        zu quittieren oder zu korrigieren.))
      (define side-effects
        '(Jeder Aufruf führt zu
         einem Eintrag im
         System-Journal))
      '|Daten Formular 412|))
;
  (define Rückrechnung
    (lambda (...)
      (define effects
        '(Ermittelt den zusätzlichen
        Anspruch bei rückwirkenden
        Änderungen der Regelsätze.))
       ...)))
```

3.2.3 Zusammenfassung: Spezifikation

Spezifizieren heißt Präzisieren und umfasst alle Phasen. Ziel ist eine Spezifikation mit der Präzision der Mathematik und der Verständlichkeit der Umgangssprache.

Die Aufgabenart (Vollzugs-, Defintions-, Zielerreichungs- oder Steuerungsproblem) und die Aufgabengröße (ein Mann, kleines Team oder große Mannschaft) erfordern unterschiedliche (rechnergestützte) Arbeitstechniken.

Eine funktionale Anforderung beschreibt eine Leistung bzw. eine quantitative oder qualitative Systemeigenschaft. Von ihr fordern wir Realisierbarkeit und Gestaltungsrelevanz. Schrittweise werden qualitative Vorgaben verfeinert, bis die einzelnen Eigenschaften durch Ersatzleistungen approximierbar sind.

Benutzt- und *Enthält*-Relationen dienen zum Spezifizieren von Moduln und ihren Wechselwirkungen. Wir definieren eine azyklische Modul-Hierarchie über die `provide`- und die `require`-Schnittstelle. Schritt

für Schritt verfeinern wir dieses „Skelett“. Dabei notieren wir Restriktionen und Nebeneffekte..

Charakteristisches Beispiel für Abschnitt 3.2

Vorgabenarten: Das System `Fürsorge` (Automation im Sozialamt einer Kommune) ist ein Definitionsproblem mit einer mittleren Konstruktionsgröße (↪ Abschnitt 60 S. 414). Dazu sind folgende Aussagen notiert:

`A1:` ;;Approximierbare Vorgabe
`Fürsorge` unterstützt ausreichend das Sozialamt-Management.

`A2:` ;;Konkurrierende Vorgabe zu `A1`
`Fürsorge` schützt den Sachbearbeiter vor permanenter Arbeitskontrolle.

`A3:` ;;Keine Vorgabe,
;; da Realisierbarkeit und Gestaltungsrelevanz fehlen.
`Fürsorge` schafft große Freude.

Die schrittweise Approximation von `A1` führt zu folgenden Aussagen:

`A1.1:` `Fürsorge` bilanziert das IST-Geschehen in Sozialamt bezogen auf Sachbearbeiter, Vorgangsart und Zeitraum.

`A1.2:` `Fürsorge` vergleicht die Bilanzergebnisse mit denen von ähnlichen Sozialämtern.

`A1.3:` `Fürsorge` stellt sogenannte „`if...then...`“-Situationen dar.

Offensichtlich konkurriert `A1.1` mit `A2`. Ein Kompromiss wäre z. B. :

`A1.1:` `Fürsorge` bilanziert das IST-Geschehen im Sozialamt bezogen auf Abteilungen, Vorgangsart alle 14 Tage.

Die Skizze einer ausgefüllten Checkliste zur Modul-Beschreibung könnte sich wie folgt darstellen:

```
eval> (module Registratur scheme
   (provide Neuzugang
            Änderung
            Löschung
            Sperrung
            Recherche)
   (require (only-in 'Journal Eintragung)
            (only-in 'Vertraulichkeit
                     Überprüfung
                     Verschlüsselung
                     Entschlüsselung))
   (define effects
```

```
     '(Speichern und Bereitstellen von Akten))
;
   (define Neuzugang
     (lambda (Aktenzeichen
              Vorgang
              Berechtigung)
       (define effects
         '(Neuzugang legt eine Akte an,
         falls unter dem Aktenzeichen
         noch keine gespeichert ist, und
         ergibt den Berechtigungscode.))
       (define constraints
         '(Zur  Datensicherheit  führt
         ein korrekter Neuzugang zum
         sofortigen Rückschreiben des
         Datensatzes in die Datei.))
       (define side-effects
         '(Das Journal weist jeden
         Aufrufmit Vertraulichkeitsstufe
         größer Klasse-V aus.))))
;
   (define Änderung
     (lambda (...) ...))
   (define Löschung
     (lambda (...) ...))
   (define Sperrung
     (lambda (...) ...))
   (define Recherche
     (lambda (...) ...)))
```

3.3 Notation mit UML

Die Qualität eines Programms hängt von der Qualität der Modelle ab, die die Objekte der Anwendungswelt sowie die (Benutzer-)Anforderungen abbilden. Erst die „richtigen“ Objekte mit den „richtigen“ Beziehungen führen zum gelungenen Programm. Kurz: *Fehler im Modell ≡ Fehler im Programm!* Insbesondere für die objekt-orientierte Analyse und den objekt-orientierten Entwurf wurden zu Beginn der 90iger-Jahre verschiedene Konzepte und Methoden mit ihren speziellen Notationen (Grafiksymbolen) entworfen.[9]

Aus solchen Methoden und Notationen hat sich als marktrelevant die *Unified Modeling Language* (UML) entwickelt. Ursprünglich vorgeschla-

[9] Exemplarisch seien genannt: G. Booch; *Objekt-oriented Analysis and Design with Applications* ↪ [15] ; D. W. Embley u. a.; *Object-Oriented Systems Analysis – A Model-Driven Approach* ↪ [58] ; I. Jacobsen u. a.; *Object-oriented Software Engineering, A Use Case Driver Approach* ↪ [99] ; W. Kim u. a.; *Object-Oriented Concepts, Databases, and Applications* ↪ [108] und J. Rumbaugh u. a.; *Object-oriented Modelling and Design* ↪ [155]

Legende:
Beispiel: `Konto%` ↪ S. 329; `Chart-Konto%`-Konstrukt ↪ S. 421

Abbildung 3.2: UML-Klassensymbol — Rechteck —

gen von der *Rational Software Corporation* (↪ [149]) und anschließend unter der Obhut der *Object Management Group* (OMG[10]) ist UML als *ISO/IEC 19501*[11] normiert worden.

UML ist eine Sprache zum *Spezifizieren*, *Visualisieren*, *Konstruieren* und *Dokumentieren* von Artefakten eines Softwaresystems. UML ist besonders gut geeignet für die Analyse von Geschäftsvorgängen (*business modeling*). Zunehmend gewinnt UML auch Bedeutung für die Modellierung von Nicht-Software-Systemen. Die große Verbreitung von UML hat zu einer umfangreichen Sammlung von besten Ingenieurpraktiken und Mustern geführt, die sich erfolgreich beim Modellieren auch sehr komplexer Systeme bewährt haben.

Der strikt objekt-orientierte UML-Standard definiert *Strukturdiagramme*, *Architekturdiagramme* und *Verhaltensdiagramme* sowie das hinter den Diagrammen stehende *Repository*.[12]

Im Mittelpunkt steht das *Klasse-Instanz*-Modell (↪ Abschnitt 2.8 S. 327) und damit das *Klassendiagramm*; ein Diagramm, das die Klassen und ihre Beziehungen untereinander zeigt. Dabei wird eine Klasse als ein Rechteck mit dem Klassennamen in der Mitte dargestellt (↪ Abbildung 3.2 S. 419). In der detaillierteren Form weist es die *Slots* (Felder) und die Methoden aus (↪ Abbildung 3.3 S. 422).

3.3.1 UML-Symbol: *Klasse%*

Einerseits um Grafik-Optionen kennenzulernen, andererseits zum weiteren Training des Verstehens von Konstrukten und Konstruktionen, befassen wir uns mit der Programmierung solcher Klassendiagramme.

Hinweis: Grafik in *PLT-Scheme*
Im Umfeld von *PLT-Scheme* gibt es eine Menge Bibliotheken zur Erzeugung

[10] OMG ↪ `http://www.omg.org/` (Zugriff: 20-Jan-2010)
[11] *International Organization for Standardization*
↪ `http://www.iso.org/iso/home.htm` (Zugriff: 20-Jan-2010)
[12] Diagramme sind also spezielle Sichten auf das *Repository* und können relativ frei gestaltet werden.

von Grafiken. Wir nutzen *Slideshow: PLT Figure and Presentation Tools*, d. h.:

```
(require slideshow)
(require slideshow/flash)
(require slideshow/code)
(require scheme/class scheme/gui/base)
```

Online-Dokumentation:
↪ `http://docs.plt-scheme.org/slideshow/index.html` (Zugriff: 21-Ja-2010)

Die bereitgestellten, elementaren grafischen Bausteine, wie z. B. eine horizontale Linie, eine vertikale Linie, ein Zirkel, ein Rechteck oder ein Text-Konstrukt, lassen sich vertikal (z. B. `vc-append`) und horizontal (z. B. `hc-append`) aneinander fügen.

`eval>` (< *prefix* >`-append` < *gap* > < *picts* >)

mit:

< *prefix* >	≡	`hc` horizontal auf *center*-Linie
		`hb` horizontal auf *base*-Linie
		`ht` horizontal auf *top*-Linie
		`vc` vertikal auf *center*-Linie
		`vl` vertikal auf *left*-Linie
		`vr` vertikal auf *right*-Linie
< *gap* >	≡	Abstand der Grafikobjekte untereinander.
< *picts* >	≡	Grafikobjekte, die `pict?` erfüllen.

Beispiel: Drei Kreise nebeneinander mit unterschiedlichen Durchmessern ($d = 50.0$; $d = 100.0$; $d = 200.0$) „schweben“ (10 Einheiten) über einer horizontalen Linie ($l = 350.0$).

```
eval> (vc-append 10
          (hb-append 0
                     (circle 50.0)
                     (circle 100.0)
                     (circle 200.0))
         (hline 350.0 0))
```

`==>` ;Bild mit Kreisen ↪ Abbildung A.1 S. 468

Das `text`-Konstrukt erzeugt aus einer Zeichenkette (`string?`) ein Grafikobjekt (`pict?`) mit Hilfe einer Instanz aus der Klasse `font%`. Dabei kann der erzeugte Text auch gedreht werden (4. Parameter). Ein Rechteck mit dem Klassennamen `Konto%` ist im folgenden `Chart-Konto%`-Konstrukt definert.

`Chart-Konto%`**-Konstrukt**

```
eval> (define Chart-Konto%
       (let ((MyFont
              (make-object font%
                40
                'modern
                'normal
                'bold)))
         (vc-append
          (hline 300.0 0.0)
          (hc-append
           55
           (vline 0.0 200)
           (text "Konto%" MyFont 50 0.0)
           (vline 0.0 200))
          (hline 300.0 0.0))))
```

`eval> Chart-Konto% ==>` ↪ Abbildung 3.2 S. 419

Die Möglichkeit `pict`-Objekte übereinander ausgeben zu können, verkürzt das `Chart-Konto%`-Konstrukt wie folgt. Das `pin-over`-Konstrukt hat als 2. und 3. Parameter (*dx*, *dy*) die Verschiebung des überlagernden Grafikobjektes (4. Parameter) gegenüber dem Bezugsobjekt (1. Parameter).

```
eval> (define Chart-Konto%
       (pin-over
        (rectangle 300 100)
        50 20
        (text
         "Konto%"
         (make-object
           font% 40 'modern 'normal 'bold)
         50 0.0)))
```

Eine abstrakte Klasse ist eine Klasse, die eine Basis für Unterklassen bildet. Eine abstrakte Klasse hat keine Instanzen (Objektexemplare). Sie wird als eine (normale) Klasse mit dem Merkmal {`abstract`} notiert[13]. Eine Metaklasse dient zum Erzeugen von Klassen. Sie wird wie eine (normale) Klasse notiert und erhält den Stereotyp ≪metaclass≫[14].

Ein Merkmal ist ein Schlüsselwort aus einer in der Regel vorgegebenen Menge, das eine charakteristische Eigenschaft benennt. Ein Merkmal steuert häufig die Quellcodegenerierung.

[13] Alternativ zu dieser Kennzeichnung kann der Klassenname auch *kursiv* dargestellt werden.

[14] Näheres zur Programmierung mit Metaobjekten ↪ z. B. [103].

Legende:
Beispiel: `Konto%` ↪ S. 329; `Chart-Konto-All%`-Konstrukt ↪ S. 423.
Gliederung in *Slots* (Felder) und Methoden.
Da die Klammern in LISP stets bedeutsam sind, wurde bei den Methoden nicht die übliche UML-Notation (primär *C++* und *Java* geprägt) verwendet, sondern „LISPisch“ notiert, also $(< method > \ < parameters >)$ statt $< method > \ (< parameters >)$.

Abbildung 3.3: UML-Klassensymbol mit Einteilung

Beispiele: {`abstract`}, {`readOnly`} oder {`old`}

Eine Zusicherung definiert eine Integritätsregel (Bedingung). Häufig beschreibt sie die zulässige Wertmenge, eine Vor- oder Nachbedingung für eine Methode, eine strukturelle Eigenschaft oder eine zeitliche Bedingung.
Beispiel: Die Kundenangabe im Vertrag zur Rechnung sei gleich der Angabe in der Rechnung.

```
(equal?
  (send Rechnung getkunde)
  (send (send Rechnung getVertrag) getKunde))
```

Die Angaben von {`zusicherung`} und {`merkmal`} überlappen sich. So kann jedes Merkmal auch als eine Zusicherung angegeben werden.
Beispiele: Merkmal als Zusicherung angegeben
{`abstract=true`}, {`readOnly=true`} oder {`old=true`}

Ein Stereotyp[15] ist eine Möglichkeit zur Kennzeichnung einer Gliederung auf projekt- oder unternehmensweiter Ebene. Ein Stereotyp gibt in der Regel den Verwendungskontext einer Klasse, Schnittstelle, Beziehung oder eines Paketes an.
Beispiele: {`fachklasse`}, {`präsentation`} oder {`vorgang`}

[15] Verwandte Begriffe für den Stereotyp sind die Begriffe Verwendungskontext und Zusicherung.

Ein Paket[16] beinhaltet eine Ansammlung von „Bausteinen“ beliebigen Typs; häufig jedoch nur Klassen. Mit Hilfe von Paketen wird ein (komplexes) Gesamtmodell in überschaubarere Einheiten gegliedert. Jeder „Baustein“ gehört genau zu einem Paket.

Wir programmieren diese UML-Notation für die Klasse `Konto%` als Konstrukt `Chart-Konto-All%`. Mit der Evaluierung dieses Konstruktes erhalten wir die Abbildung 3.3 S. 422.

`Chart-Konto-All%`**-Konstrukt**

Hier nutzen wir das `frame`-Konstrukt um das Rechteck abzubilden und definieren für die verschiedenen Texte passende Fonts.

```
eval> (define Chart-Konto-All%
  (let ((fontB
         (make-object font%
           40
           'modern
           'normal
           'bold))
        (fontI
         (make-object font%
           20
           'modern
           'italic
           'normal))
        (fontN
         (make-object font%
           20
           'modern
           'normal
           'normal))
        (width 850.0))
    (frame
     (vc-append
      (text "<<stereotyp>>" fontI 30 0.0)
      (hc-append
       0
       (text "paket::" fontN 50 0.0)
       (text "Konto%" fontB 50 0.0))
      (text "{Merkmal}" fontI 30 0.0)
      (hline width 20.0)
      (text
       "variable: Typ=initialwert {merkmal}{zusicherung}"
       fontN 30 0.0)
      (hline width 20.0)
      (text
       "(methode param: Typ=initialwert) :              "
       fontN 30 0.0)
      (text
```

[16] Verwandte Begriffe für das Paket sind die Begriffe Klassenkategorie, Subsystem und *Package*.

```
       "             Rückgabetyp {merkmal} {zusicherung}"
       fontN 30 0.0)))))
```

`eval> Chart-Konto-All% ==>` ↪ Abbildung 3.3 S. 422

Mit den definierten *Slots* und Methoden unseres Beispiels `Konto%` (↪ S. 329) ergibt sich die Abbildung 3.4 S. 427. Diese Abbildung erzeugen wir mit dem folgenden Konstrukt `Chart-Konto-Example%`.

`Chart-Konto-Example%`**-Konstrukt**

```
eval> (define Chart-Konto-Example%
  (let ((fontB
         (make-object font%
           40
           'modern
           'normal
           'bold))
        (fontI
         (make-object font%
           20
           'modern
           'italic
           'normal))
        (fontN
         (make-object font%
           20
           'modern
           'normal
           'normal))
        (width 850.0))
    (frame
     (vc-append
      (text "<<fachklasse>>" fontI 30 0.0)
      (hc-append
       0
       (text "HEGB::" fontN 50 0.0)
       (text "Konto%" fontB 50 0.0))
      (text "{public, version=1.0}" fontI 30 0.0)
      (hline width 20.0)
      (text
       "+Identifizierung : number?=id              "
       fontN 30 0.0)
      (text
       "+Name-Inhaber : string?=name               "
       fontN 30 0.0)
      (text
       "+Adresse-Inhaber : string?                 "
```

```
 fontN 30 0.0)
(text
 "-Einzahlungen : list?                        "
 fontN 30 0.0)
(text
 "-Auszahlungen : list?                        "
 fontN 30 0.0)
(text
 "-Kontostand : number?=0                      "
 fontN 30 0.0)
(hline width 20.0)
(text "<<Selektoren>>" fontI 30 0.0)
(text
 "+(get-Identifizierung) :  number?            "
 fontN 30 0.0)
(text
 "+(get-Name-Inhaber) :  string?               "
 fontN 30 0.0)
(text
 "+(get-Adresse-Inhaber) :  string?            "
 fontN 30 0.0)
(text
 "+(get-Einzahlungen) :  list?                 "
 fontN 30 0.0)
(text
 "+(get-Auszahlungen) :  list?                 "
 fontN 30 0.0)
(text
 "+(get-Kontostand) :  number?                 "
 fontN 30 0.0)
(text "<<Mutatoren>>" fontI 30 0.0)
(text
 "+(set-Name-Inhaber! Name : string?)          "
 fontN 30 0.0)
(text
 "+(set-Adresse-Inhaber! Adresse : string?)    "
 fontN 30 0.0)
(text
 "+(set-Einzahlungen! Wert : list?)            "
 fontN 30 0.0)
(text
 "+(set-Auszahlungen! Wert : list?)            "
 fontN 30 0.0)
(text
 "-(set-Kontostand! Wert : number?)            "
 fontN 30 0.0)
(text "<<Fachspezifische Methoden>>"
 fontI 30 0.0)
(text
 "+(Add-Einzahlungen! E_Datum  Betrag)         "
 fontN 30 0.0)
(text
 "+(Add-Auszahlungen! E_Datum  Betrag)         "
 fontN 30 0.0)))))
```

```
eval> Chart-Konto-Example% ==> ↪ Abbilung 3.4 S. 427
```

3.3.2 Assoziation und Vererbung

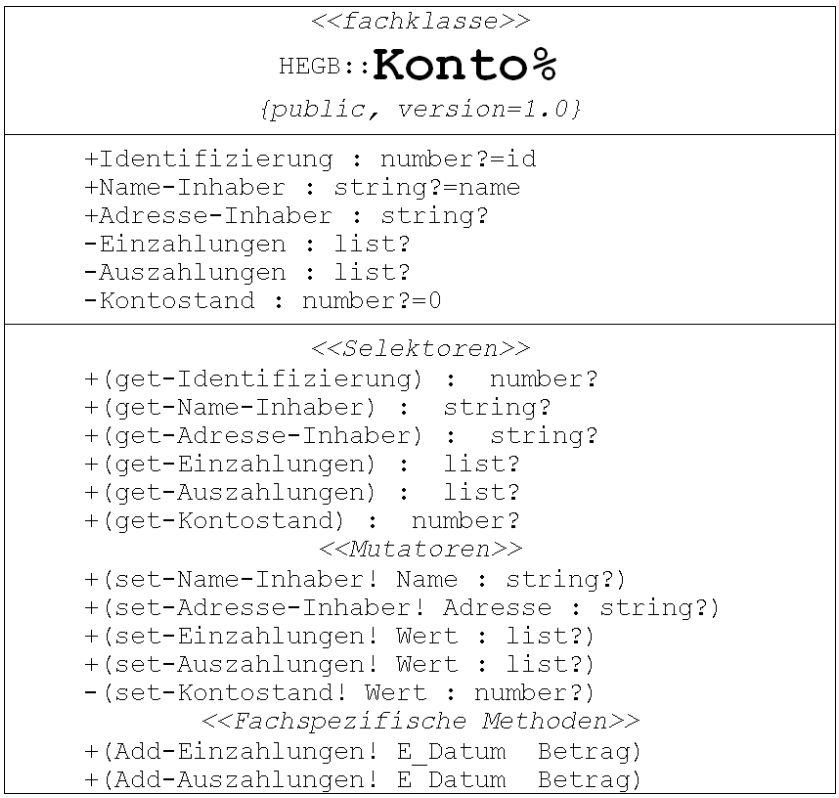
Das UML-Klassendiagramm dokumeniert die Beziehungen zwischen einzelnen Klassen. Bedeutsam sind die Beziehungen *Assoziation* und *Vererbung*. Im Folgenden werden beide erläutert und ihre grafische Darstellung programmiert.

Assoziation

Eine Assoziation wird in UML als eine (gerichtete) Verbindungslinie zwischen Klassen dargestellt (↪ Abbildung 3.5 S. 428). Die Beziehung zwischen einer Instanz der einen Klasse mit einer Instanz der „anderen“ Klasse wird Objektverbindung (englisch: *link*) genannt. Links lassen sich daher als Instanzen einer Assoziation auffassen.

Exemplarisch für eine Assoziation betrachten wir die Beziehung zwischen der Klasse `Foo%`, die ein Feld `slot` über die Zugriffsmethoden `get-Slot` und und `set-Slot!` bereitstellt und der Klasse `Bar%`, die die Methode `add1` definiert. Speichern wir nun in einer erzeugten Instanz (`instanceFoo`) der Klasse `Foo%` im Feld `slot` eine anonyme Instanz der Klasse `Bar%`, dann besteht eine Assoziation zwischen den Klassen. Im UML-Klassendiagramm wird die Assoziation durch eine Linie zwischen den beiden Klassen dargestellt.

```
eval> (define Foo%
       (class object%
        (super-new)
        (define slot "OK!")
        (define/public get-Slot
         (lambda () slot))
        (define/public set-Slot!
         (lambda (Slot)
          (set! slot Slot)))))
eval> (define Bar%
       (class object%
        (super-new)
         (define/public add1
          (lambda (n) (+ n 1)))))
eval> (define instanceFoo (new Foo%))
eval> (send instanceFoo
            set-Slot! (new Bar%))
eval> (send instanceFoo get-Slot)
      ==> #(struct:object:Bar% ...)
eval> (send
```

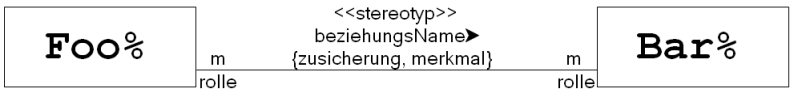


Legende:

+	allgemein verfügbares Konstrukt (*Slot* / Methode)
−	privates, nicht allgemein zugreifbares Konstrukt
< *name* >	Klassenkonstrukt (*Slot* / Methode)

Beispiel: `Konto%` ↪ S. 329; `Chart-Konto-Example%`-Konstrukt ↪ S. 424.

Abbildung 3.4: UML-Klassenbeispiel: `Konto%`



Legende:

beziehungsName	Name der Assoziation
{zusicherung}	Bedingung, die einzuhalten ist.
{merkmal}	besonderes Kennzeichen
m	Multiplizität, z. B. $*$ oder 0..3
rolle	Sichtweise durch das gegenüberliegende Objekt
$\triangleright$	(Lese)-Richtung für die Beziehung; hier: `Foo% beziehungsName Bar%`

Abbildung 3.5: UML-Beziehungselement: Assoziation

```
(send instanceFoo get-Slot)
 add1 6)  ==> 7
```

Die Assoziation selbst kann einen Namen und eine Beziehungsrichtung („Leserichtung“) haben. Die Anzahl der Instanzen, die in der assoziierten Instanz vorkommen können, sind durch die Angabe der Multiplizität über der Verbindungslinie darstellbar. Unterhalb der Linie kann die jeweilige Rolle der Instanzen bei der Assoziation vermerkt werden. Die Multiplizität (*m* in Abbildung 3.5 S. 428) gibt an, mit wievielen Instanzen der gegenüberliegende Klasse `Bar%` eine Instanz der Klasse (`Foo%`) assoziiert ist[17]. Dabei kann eine Bandbreite durch den Mini- und den Maximumwert angegeben werden.

Eine Assoziation wird in der Regel so implementiert, dass die beteiligten Klassen zusätzlich entsprechende Referenzslots bekommen. Wir nehmen beispielhaft die Aussage an: *„Ein Unternehmen beschäftigt viele Mitarbeiter“*. Die Klasse `Unternehmen%` enthält dann den Slot `arbeitnehmer` und die Klasse `Mitarbeiter%` den Slot `arbeitgeber`. Aufgrund der angebenen Multiplizität, hier sei $m = *$ („viele Mitarbeiter“), muss der Slot `arbeitnehmer` mehrere Werte aufnehmen, d. h. einer „Behälterklasse“ entsprechen. Ein *Set* ($\equiv$ Menge ohne Duplizität) oder ein *Bag* ($\equiv$ Menge mit Duplizität) sind beispielsweise übliche Behälterklassen. In der Regel wird dann der jeweilige Rollenname; hier `rolle`, für den Referenzslot verwendet.

Soll eine Assoziation eine Bedingung erfüllen, dann ist diese in Form der Zusicherung (`{zusicherung}`) neben der Assoziationslinie zu notieren. Eine Zusicherung kann auch die referenzielle Integrität beschreiben. Z. B. könnten hierzu beim Löschen angebenen werden:

- `{prohibit deletion}`

[17] ... beziehungsweise assoziiert sein kann.

Das Löschen eines Objektes ist nur erlaubt, wenn keine Beziehung zu einem anderen Objekt besteht.

- {delete link}
  Wenn ein Objekt gelöscht wird, dann wird nur die Beziehung zwischen den Objekten gelöscht.
- {delete related object}
  Wenn ein Objekt gelöscht wird, dann wird das assoziierte („gegenüberliegende“) Objekt ebenfalls gelöscht.

`Chart-Association%`**-Konstrukt**

```
eval> (define Chart-Association%
  (let ((MyFont
         (make-object font%
           40
           'modern
           'normal
           'bold)))
    (hc-append
     (pin-over
      (rectangle 240 100)
      50
      20
      (text "Foo%" MyFont 50 0.0))
     (vc-append
      (text "<<stereotyp>>" null 24 0.0)
      (hc-append
       (text "beziehungsName" null 24 0.0)
       (arrowhead 16 0.0))
      (hc-append
       80
       (text "m" null 24 0.0)
       (text "{zusicherung, merkmal}" null 24 0.0)
       (text " m" null 24 0.0))
      (hline 500 1)
      (hc-append
       400
       (text "rolle" null 24 0.0)
       (text "rolle" null 24 0.0)))
     (pin-over
      (rectangle 240 100)
      50
      20
      (text "Bar%" MyFont 50 0.0)))))
```

`eval> Chart-Association% ==>` ↪ Abbildung 3.5 S. 428

Assoziation — Sonderfall Aggregation

Eine Aggregation beschreibt eine „Ganzes⇔Teile“-Assoziation. Das Ganze nimmt dabei Aufgaben stellvertretend für seine Teile wahr. Im Unterschied zur normalen Assoziation haben die beteiligten Klassen keine gleichberechtigten Beziehungen. Die Aggregationsklasse hat eine hervorgehobene Rolle und übernimmt die „Koordination“ ihrer Teileklassen. Zur Unterscheidung zwischen Aggregationsklasse und Teileklasse(n) wird die Beziehungslinie durch eine Raute (◇) auf der Seite der Aggregationsklasse ergänzt. Die Raute symbolisiert das Behälterobjekt, das die Teile aufnimmt.

Vererbung

Im Abschnitt 2.8.2 S. 337 wurde die Klasse `Konto%` durch eine Klasse `Verwahrkonto%` spezialisiert. Anders formuliert: die Klasse `Konto%` vererbt ihre Eigenschaften (*Slots* und Methoden) an die Klasse `Verwahrkonto%`. Instanzen dieser Klasse haben beide Eigenschaften, die ihrer Superklasse und ihrer eigenen Klasse. Diese Vererbung (*Inheritance*) wurde wie folgt konstruiert:

```
eval> (define Verwahrkonto% (class Konto% ...)
```

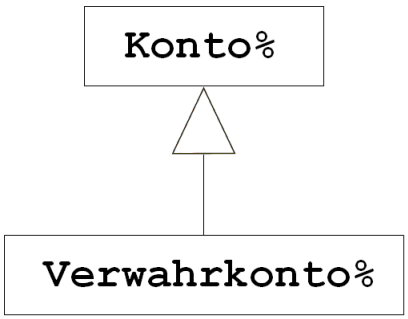
In UML wird diese Beziehung zwischen den Klassen durch einen Pfeil mit einer besonders großen Spitze (△) von der Unterklasse zur Oberklasse abgebildet. Diese Pfeilspitze lässt sich als Bild mit dem `bitmap`-Konstrukt in die Zeichung einbauen. Die Bilddatei ist über das Konstrukt `build-path` zu spezifizieren. Dabei kommen das `gif`-Format[18] und das `jpg`-Format[19] in Betracht.

`Chart-Inheritance%`-Konstrukt

```
eval> (define Chart-Inheritance%
       (let ((MyFont
              (make-object font%
                40
                'modern
                'normal
                'bold)))
         (vc-append
```

[18] *GIF* (*Graphics Interchange Format*) ist ein Bildformat mit verlustfreier Komprimierung für Bilder mit geringer Farbtiefe.
↪ `http://www.w3.org/Graphics/GIF/spec-gif89a.txt` (Zugriff: 18-Jan-2010)

[19] *JPEG*, bzw. *JPG*, ist die umgangsprachliche Bezeichnung für die Norm *ISO/IEC 10918-1*, die von der *Joint Photographic Experts Group* Anfang der 90iger Jahre entwickelt wurde.
↪ `http://www.jpeg.org/` (Zugriff: 18-Jan-2010)



Legende:
Beispiel: `Verwahrkonto%` ↪ S. 337; `Chart-Konto-Example%`-Konstrukt ↪ S. 430.

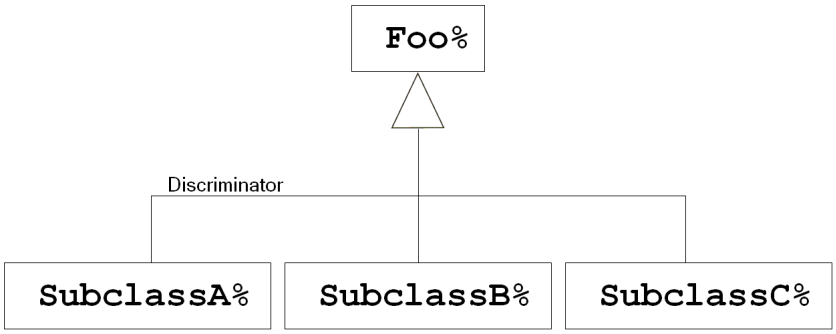
Abbildung 3.6: UML-Vererbungsbeispiel: `Verwahrkonto%`

```
(pin-over
 (rectangle 300 100)
  50
  20
(text "Konto%" MyFont 50 0.0))
(bitmap
 (build-path
  "D:\\bonin\\scheme\\image"
  "Vererbungspfeil.gif"))
(filled-rectangle 1 100)
(pin-over
 (rectangle 500 100)
 50
 20
 (text "Verwahrkonto%"
   MyFont 50 0.0)))))
```

`eval> Chart-Inheritance% ==>` ↪ Abbilung 3.6 S. 431

Mit der Vererbung wird eine Klassenhierarchie aufgebaut. Welche gemeinsamen Eigenschaften von Unterklassen zu einer Oberklasse zusammengefasst, also generalisiert werden und umgekehrt, welche Eigenschaften der Oberklasse in Unterklassen genauer beschrieben, also spezialisiert werden, ist abhängig vom jeweiligen charakteristischen Unterscheidungsmerkmal der einzelnen Unterklassen.

Ein solches Merkmal wird *Diskriminator* genannt (↪ Abbildung 3.7 S. 432). Er ist das charakteristische Gliederungsmerkmal für die Zuordung in eine Oberklasse oder eine Unterklasse. Er bestimmt einerseits die Generalisierung und andererseits die Spezialisierung. Spezialisiert



Legende:

△	≡	Vererbungsrichtung
Dicriminator	≡	charakteristisches Gliederungsmerkmal für die **Generalisierungs⇔Spezialisierungs**-Beziehung
Foo%	≡	Oberklasse von SubclassA%, SubclassB% und SubclassC% — Eigenschaften von Foo% sind in SubclassA..C% zugreifbar.

Chart-Discriminator%-Konstrukt ↪ S. 432.

Abbildung 3.7: UML: Generalisierung & Spezialisierung

man z. B. eine Klasse Schiff% in die Klassen Containerschiff%, Tanker% und Autotransporter%, dann ist der Diskriminator die Ladungsart.

Das folgende Konstrukt Chart-Discriminator% nutzt die primitiven Grafik-Komponenten hline, vline, rectangle, text und bitmap. Das Zusammenfügen dieser primitiven Grafik-Komponenten erfolgt in vertikaler Richtung mit vc-append und in horizontaler mit hb--append und ht-append. Das pin-over-Konstrukt ermöglicht Grafik-Komponenten übereinander zu zeichen. Mit blank wird die Positionierung beim Zusammenfügen beeinflusst.

Chart-Discriminator%-Konstrukt

```
eval> (define Chart-Discriminator%
  (let ((MyFont
         (make-object font%
           40
           'modern
           'normal
           'bold)))
    (vc-append
     (pin-over
      (rectangle 200 100)
      50
      20
```

```
 (text "Foo%" MyFont 50 0.0))
(bitmap
 (build-path
  "D:\\bonin\\scheme\\image"
  "Vererbungspfeil.gif"))
(hb-append
 200
 (text "Discriminator" null 30 0.0)
 (vline 0 100)
 (blank 175 0))
(hline 800 0)
(ht-append
 400
 (vline 0 100)
 (vline 0 100)
 (vline 0 100))
(ht-append
 20
 (pin-over
  (rectangle 400 100)
  50
  20
  (text "SubclassA%" MyFont 50 0.0))
 (pin-over
  (rectangle 400 100)
  50
  20
  (text "SubclassB%" MyFont 50 0.0))
 (pin-over
  (rectangle 400 100)
  50
  20
  (text "SubclassC%" MyFont 50 0.0))))))
```

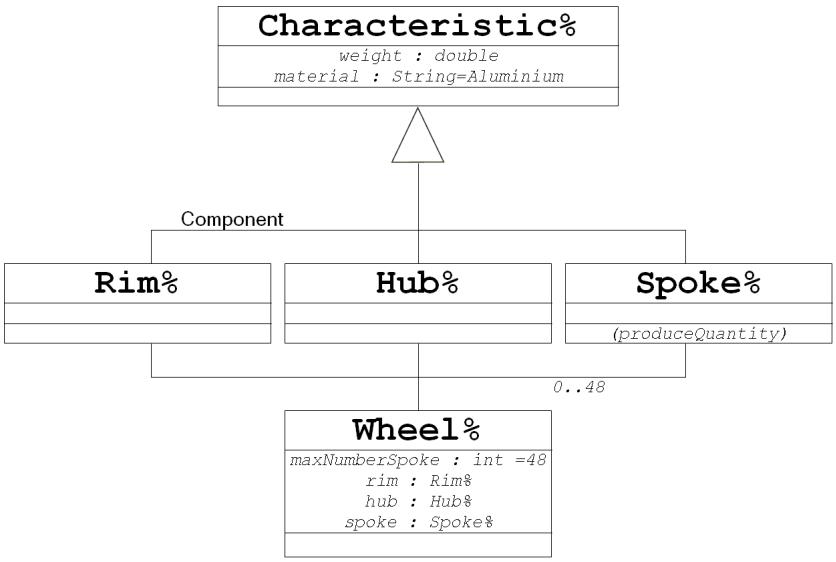
`eval> Chart-Discriminator% ==>` ↪ Abbildung 3.7 S. 432

3.3.3 Zusammenfassung: UML

Mit vielen Rechtecken (Boxen), die mit Linien und Pfeilen verbunden sind, kann ein *Klasse-Instanz*-Modell dargestellt werden. Holzschnittartig formuliert, die „UML-Boxologie“[20] leistet einen nützlichen Beitrag für die Durchschaubarkeit (Transparenz) von objekt-orientierten Systemen.

Die simple UML-Grafik lässt sich leicht mit LISP-Konstrukten programmieren. Zum Training und zur Demonstration der Grafikoptionen

[20] Angemerkt sei, dass diese Boxologie nichts mit den Boxern im Sinne von Faustkämpfern oder im Sinne des chinesischen Geheimbundes um 1900 zu tun hat. Auch die Box als eine einfache Kamera oder als Pferdeunterstand sind hier keine hilfreiche Assoziation.



Legende:
`Chart-Wheel%`-Konstrukt ↪ S. 434

Abbildung 3.8: UML: Klasse `Wheel%`

wurde exemplarisch das UML-Klassendiagramm programmiert. Hier ist die folgende Aussage als ein UML-Klassendiagramm (↪ Abbildung 3.8 S. 434) entsprechend programmiert:

> *„Ein Laufrad besteht aus einer Felge, einer Nabe und bis zu 48 Speichen. Relevant sind Gewicht und Material für alle Teile.“*

`Chart-Wheel%`-Konstrukt

```
;;;;Beispiel: UML-Diagram Wheel%
;;;;
(module Chart-Wheel% scheme
  (require slideshow)
  (require slideshow/flash)
  (require slideshow/code)
  (require scheme/class scheme/gui/base)
  (provide Chart-Wheel%)
  (define Chart-Wheel%
    (lambda ()
      (let ((fontBig
             (make-object font%
               40
```

```
        'modern
        'normal
        'bold))
     (fontSmall
      (make-object font%
        20
        'modern
        'italic
        'normal)))
(vc-append
 (pin-over
  (rectangle 600 150)
  0
  0
  (vc-append
   (text "Characteristic%" fontBig 50 0.0)
   (hline 600 0)
   (text
    "weight : double"
    fontSmall
    20
    0.0)
   (text
    "material : String=Aluminium"
    fontSmall
    20
    0.0)
   (hline 600 0)))
 (bitmap
  (build-path
   "D:\\bonin\\scheme\\image"
   "Vererbungspfeil.gif"))
 (hb-append
  200
  (text "Component" null 30 0.0)
  (vline 0 100)
  (blank 155 0))
 (hline 800 0)
 (ht-append
  400
  (vline 0 50)
  (vline 0 50)
  (vline 0 50))
 (ht-append
  20
  (pin-over
   (rectangle 400 120)
   0
   0
   (vc-append
    (text "Rim%" fontBig 50 0.0)
    (hline 400 0)
    (text "" fontSmall 20 0.0)
    (hline 400 0)))
  (pin-over
   (rectangle 400 120)
```

```
  0
  0
  (vc-append
   (text "Hub%" fontBig 50 0.0)
   (hline 400 0)
   (text "" fontSmall 20 0.0)
   (hline 400 0)))
 (pin-over
  (rectangle 400 120)
  0
  0
  (vc-append
   (text "Spoke%" fontBig 50 0.0)
   (hline 400 0)
   (text "" fontSmall 20 0.0)
   (hline 400 0)
   (text
    "(produceQuantity)"
    fontSmall
    20
    0.0))))
(ht-append
 400
 (vline 0 50)
 (vline 0 50)
 (vline 0 50))
(hline 800 0)
(ht-append
 200
 (blank 80 0)
 (vline 0 50)
 (text "0..48" fontSmall 20 0.0))
(pin-over
 (rectangle 400 220)
 0
 0
 (vc-append
  (text "Wheel%" fontBig 50 0.0)
  (hline 400 0)
  (text
   "maxNumberSpoke : int =48"
   fontSmall
   20
   0.0)
  (text "rim : Rim%" fontSmall 20 0.0)
  (text "hub : Hub%" fontSmall 20 0.0)
  (text
   "spoke : Spoke%"
   fontSmall
   20
   0.0)
  (hline 400 0)
  (text " " fontSmall 20 0.0))))))))
```

Kommando — Tastenkombination	**Aktion**
Alt-q, Meta-q, or Cmd-q	*end slide show*
Esc	*if confirmed, end show*
Right arrow, Space, f, n, or click	*next slide*
Left arrow, Backspace, Delete, or b	*previous slide*
g	*last slide*
1	*first slide*
Alt-g, Cmd-g, or Meta-g	*select a slide*
Alt-p, Cmd-p, or Meta-p	*show/hide slide number*
Alt-c, Cmd-c, or Meta-c	*show/hide commentary*
Alt-d, Cmd-d, or Meta-d	*show/hide preview*
Alt-m, Cmd-m, or Meta-m	*show/hide mouse cursor*
Shift with arrow	*move window 1 pixel*
Alt, Meta, or Cmd with arrow	*move window 10 pixels*

Legende:
Kommandos zum Navigieren in der *Slideshow*; Quelle:
↪`http://docs.plt-scheme.org/slideshow/Creating_Slide_Presentations.html`
(Zugriff: 21-Jan-2010).

Tabelle 3.11: Slideshow — Kommandos —

```
eval> (require 'Chart-Wheel%)
eval> Chart-Wheel% ==> ↪ Abbildung 3.8 S. 434
```

3.4 Präsentation — Slideshow

Die im Abschnitt 3.3.1 S. 419 gezeigten Grafik-Optionen nutzen und erweitern wir zum Aufbau einer Präsentation von „Folien“ (*Slides*). Anders als bei dem in der Regel verwendeten *Microsoft Powerpoint*, das primär auf dem *WYSIWYG*-Konzept[21] basiert, generieren wir die einzelnen Bilder und deren Präsentationsfolge durch ein *PLT-Scheme*-Programm.[22]

Eine *Slideshow* basiert auf einer Sequenz von `slide`-Konstrukten, die Texte, Bilder und zu evaluierenden LISP-Konstrukte enthalten können. Das Präsentieren mit Vor- und Rückwärtsblättern erfolgt über „Tasten-Kommandos“. Die Tabelle 3.11 S. 437 zeigt die Kommandos, die zum Navigieren bei der *Slideshow* zur Verfügung stehen.

Einzelne Ausgaben in einem *Slide*, die erst nach einem Klick dargestellt werden sollen, werden mit einem vorangestellten `'next`-Konstrukt markiert.

[21] *WYSIWYG* ≡ *What You See Is What You Get*
[22] Analog zu *SliTex*, das auf TEX-Code basiert.
↪ `http://tug.ctan.org` (Zugriff: 18-Jan-2009)

Ein charakteristisches Merkmal ist die Option, innerhalb des `slide`-Konstruktes LISP-Code ausführen zu können. Das folgende spaßige Kurzbeispiel `Lotto` zeigt Ihre Glückzahlen, indem es jeweils das `ramdom`-Konstrukt evaluiert.

Modul `Lotto`

```
(module Lotto scheme
 (require slideshow)
 (define Zahl
   (lambda ()
     (number->string
       (+ (random 48) 1))))
 (slide
  (item "Ihre Lotto Glückszahlen:")
  (item (Zahl) (Zahl) (Zahl)
        (Zahl) (Zahl) (Zahl))))
```

`eval> (require 'Lotto) ==>` Ein *Slide* mit Text und 6 Zahlen

3.4.1 Plädoyer für LISt Processing

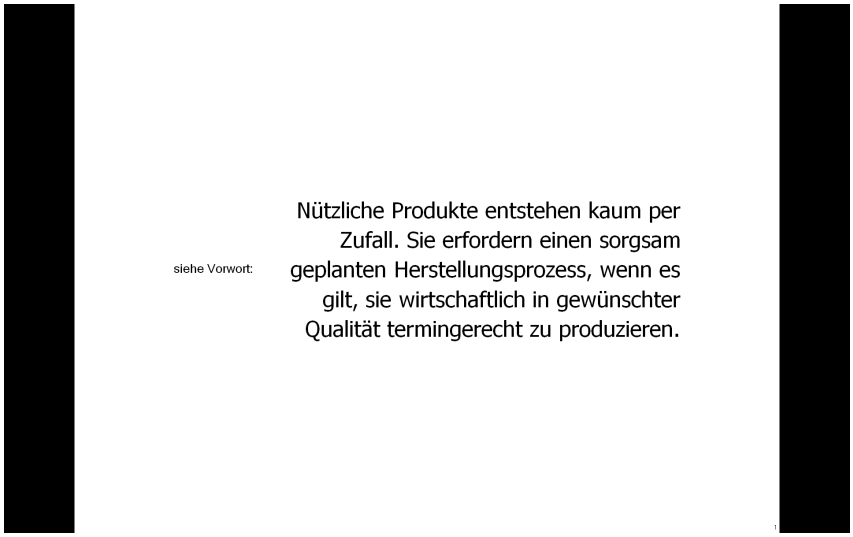
Wir nehmen nun an, wir hätten die Aufgabe, Programmierern, die mit anderen Sprachen arbeiten (z. B. mit $C++$ oder *Java*), in einem Kurzvortrag für LISP zu begeistern. Unser *Storyboard*[23] skizziert dazu 7 *Slides*, z. B. mit der Hervorhebung der LISP-spezifischen Dualität von Daten und Programm.

Für die *Slides* dieses Plädoyers definieren wir eine einheitliche Titelzeile mit Überschrift, der jeweiligen *Slide*-Nummer, dem Datum mit Uhrzeit und unserem Logo. Letzteres befindet sich als `gif`-Datei (↪ S. 430) in einem besonderen `image`-Pfad.

Das Datum mit Uhrzeit lässt sich entsprechend dem folgenden Konstrukt `Datum` definieren. Dabei wird das Datum in der für Deutschland üblichen Form, mit Hilfe des Konstruktes `date-display-format`, ausgegeben.

```
eval> (require scheme/date)
eval> (define Datum
        (lambda ()
         (date-display-format
          'german)
         (date->string
```

[23] *Storyboard* ≡ Visualisierung eines Konzeptes bzw. einer Idee; häufig eine zeichnerische Drehbuchversion.



Legende:
Beispiel: `Zitat` ↪ S. 439

Abbildung 3.9: Slide: Zitat

```
          (seconds->date
          (current-seconds))
        ;Mit Uhrzeit
        #t)))
eval> (Datum) ==>
      "21. Januar 2010, 13.06"
```

Die schon im Abschnitt 3.3 S. 418 erläuterten Konstrukte werden bei der `LISP-Slideshow` genutzt. Darüber hinaus verwenden wir das `para`-Konstrukt. Es definiert einen Absatz (*Paragraph*), der seinen Text mit Umbruch darstellt. Das folgende Beispiel `Zitat` verdeutlicht dieses Konstrukt. Zusätzlich nutzt es noch die Kommentierungsmöglichkeit mit Hilfe des `comment`-Konstruktes.

Modul `Zitat`

```
eval> (module Zitat scheme
      (require slideshow)
      (slide
       (hc-append 20
        (text "siehe Vorwort:" null 18 0)
        (para
```

```
          #:width 600
          #:align 'right
          "Nützliche Produkte entstehen"
          "kaum per Zufall. Sie erfordern einen"
          "sorgsam geplanten Herstellungsprozess,"
          "wenn es gilt, sie wirtschaftlich"
          "in gewünschter Qualität"
          "termingerecht zu produzieren."))
       (comment "Software = Produkt")))
```

```
eval> (require 'Zitat) ==> ↪ Abbildung 3.9 S. 439
```

Um die Ausführung einer Berechnung im `slide`-Konstrukt zu zeigen, definieren wir die Fakultätsfunktion („!“), das klassische Beispiel einer Rekursion:

```
eval> (define !
       (lambda (n)
        (cond ((= n 0) 1)
              (#t (* n
                   (!
                    (- n 1)))))))
eval> (! 10) ==> 3628800
```

Nun lässt sich mit dem (fiktiven) *Storyboard* folgende *Slideshow* definieren.

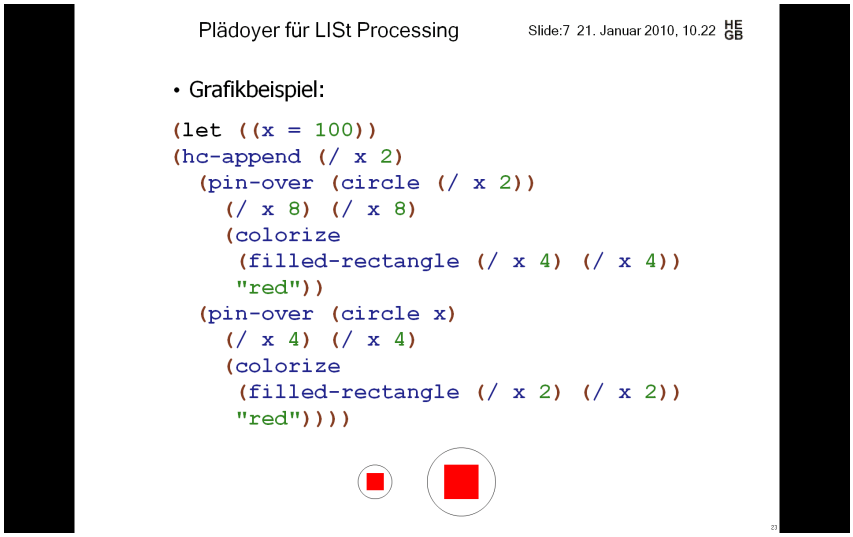
Modul `LISP-Slideshow`

```
;;;;Ein Plädoyer für LISt Processing
;;;; Version 1.0
(module LISP-Slideshow scheme
 (require slideshow)
 (require slideshow/flash)
 (require slideshow/code)
 (require scheme/class scheme/gui/base)
 (require scheme/date)

;;Fakultätsfunktion wird im Slide aufgerufen
(define !
  (lambda (n)
   (cond ((= n 0) 1)
         (#t (* n (! (- n 1)))))))
;;Deutsches Datum mit Uhrzeit
(define Datum
  (lambda ()
   (date-display-format 'german)
   (date->string
```

```
    (seconds->date (current-seconds)) #t)))
;;Mein Normalfont
(define font40nmb
  (make-object font%
           40
           'modern
           'normal
           'bold))
;;Initialisierung eines Slide-Zählers
(define Slide-Anzahl 0)
;;Titel mit Datum, Uhrzeit und Logo
(define myTitle
      (lambda ()
       (set! Slide-Anzahl
             (+ Slide-Anzahl 1))
     (hc-append 100
     (blank 30 10)
     (text
       "Plädoyer für LISt Processing"
       null 30 0)
     (hc-append 10
     (text
       (string-append "Slide:"
        (number->string Slide-Anzahl))
       null 20 0)
     (text (Datum) null 20 0)
     (bitmap
       (build-path
       "D:\\bonin\\scheme\\image"
       "hegb.gif"))))))

(slide #:title (myTitle)
      (text
        "LISP-Daten = LISP-Programm"
        font40nmb 10 0)
      'next
      (text
        "LISP-Programm = LISP-Daten"
        font40nmb 10 0)
      'next
      (text
        "Folge: Mächtiges Werkzeug"
        font40nmb 10 0)
      'next
      (colorize
        (text "erfordert Experten!" null 60 0.3)
        "red"))
(slide
   #:title (myTitle)
  (para "Programmieren ist ein systematischer,"
        "disziplinierter Konstruktionsvorgang.")
  'next
  (para "LISP passt zu allen Phasen"
        "des Konstruktionsvorganges.")
  'next
  (para "LISP erleichtert die "
```

```
        "Bewältigung komplexer Aufgaben"
        "und ist prädestiniert für den"
        "Umgang mit Symbolen."))
(slide
  #:title (myTitle)
  (para "LISP ist abgeleitet aus dem"
        "lambda-Kalkül und"
        "geprägt durch die Dualität"
        "von Daten und Programm.")
  (para "LISP ermöglicht mit seinem"
        "modernen Dialekt"
        (frame
          (colorize
            (text  "Scheme"
              font40nmb 10 0) "red"))
        "ohne Ballast:")
  'next
  (para
    (colorize
      (text  "imperativ-geprägtes"
        font40nmb 10 0) "red")
    "Programmieren")
   'next
  (para
    (colorize
      (text "funktions-geprägtes"
        font40nmb 10 0) "red")
    "Programmieren")
   'next
  (para
    (colorize
      (text "objekt-geprägtes"
        font40nmb 10 0) "red")
     "Programmieren"))

(slide
  #:title (myTitle)
 (para "Ein systematischer"
       "Konstruktionsvorgang bedingt:")
  'next
 (item "Zweckmäßige Bausteine (Konstrukte)")
 'next
 (item "Vielfältige (Muster-) Konstruktionen")
 'next
 (item "Bewährte Konstruktionsempfehlungen")
 'next
 (colorize
   (text
     "LISP erfüllt diese Bedingungen."
     font40nmb 10 0) "red"))

(slide
  #:title (myTitle)
  (para "Das übliche erste Beispiel in"
         (colorize (text  "Scheme" font40nmb 10 0)
           "red") ":")
```

```
  (code (module Start scheme
   ((lambda () 'Hello_World!)))))

(slide
  #:title (myTitle)
 (para "Klassische Rekursionsbeispiel:"
       "Fakultätsfunktion" )
  'next
  (code (define !
    (lambda (n)
      (cond ((= n 0) 1)
            (#t (* n (! (- n 1))))))))
  'next
    (item "(! 10) ==>")
    'next
   (text (number->string (! 10))
     font40nmb 10 0))

(slide
  #:title (myTitle)
   (item "Grafikbeispiel:")
   (code
   (let ((x = 100))
   (hc-append (/ x 2)
     (pin-over (circle (/ x 2))
       (/ x 8) (/ x 8)
       (colorize
        (filled-rectangle (/ x 4) (/ x 4))
        "red"))
     (pin-over (circle x)
       (/ x 4) (/ x 4)
       (colorize
        (filled-rectangle (/ x 2) (/ x 2))
        "red")))))
    'next
    (let ((x 100))
    (hc-append (/ x 2)
     (pin-over (circle (/ x 2)) (/ x 8) (/ x 8)
       (colorize (filled-rectangle (/ x 4) (/ x 4))
         "red"))
     (pin-over (circle x) (/ x 4) (/ x 4)
       (colorize (filled-rectangle (/ x 2) (/ x 2))
         "red")))))
(slide
  #:title (myTitle)
   (item "Wunsch von:")
   (code
    (list "Hinrich E. G. Bonin"
     (quote ((V)erfasser))))
    'next
    (para "Tschüß und viel Freude"
          "mit tausenden von Klammern!"))
) ; Ende module
;;;Ende LISP Slideshow
```



Legende:
Beispiel: `Verwahrkonto%` ↪ S. 337;
`LISP-Slideshow`-Konstrukt ↪ S. 440.

Abbildung 3.10: LISP-Slideshow — Slide 7 —

`eval> (require 'LISP-Slideshow) ==>` ↪ Abbildung 3.10 S. 444

3.4.2 Zusammenfassung: *Slideshow*

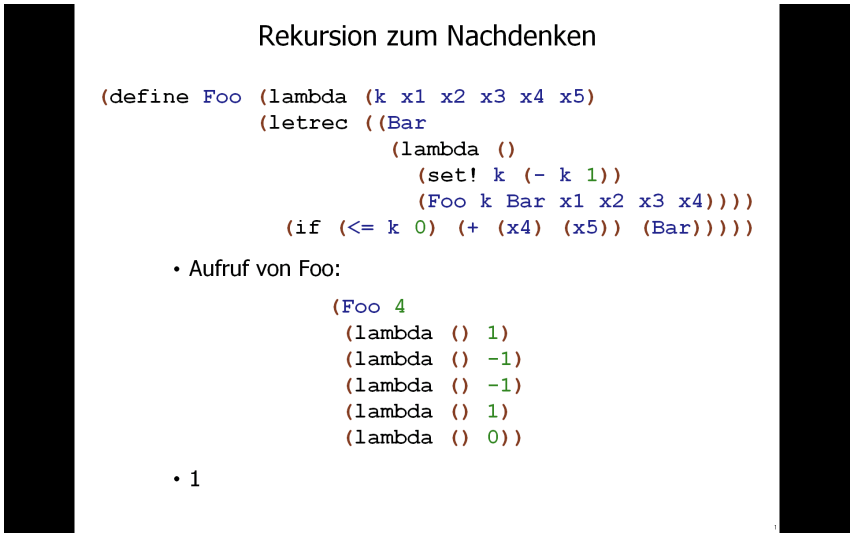
Mit der in *PLT-Scheme* gezeigten Option eine *Slideshow* zu programmieren, wird deutlich, das LISP in allen Phasen des Lebenszyklus einer Software (↪ Abbildung 1.13 S. 51) vorteilhaft eingesetzt werden kann: Von der Spezifikation (↪ Abschnitt 3.2 S. 403) bis zur Präsentaion von Dokumenten.

Charakteristisches Beispiel für Abschnitt 3.4

```
;;;;Idee von Hartmut Fritzsche/ Immo O. Kerner;
;;;; Man or Boy? Probleme der Rekursion
;;;; in Compilern,
;;;; in: Informatik-Spektrum 20: 151-158 (1997)
(module LISP-Wizard scheme
```

```
  (require slideshow)
  (require slideshow/code)
  (define Foo
    (lambda (k x1 x2 x3 x4 x5)
      (letrec ((Bar
                (lambda ()
                  (set! k (- k 1))
                  (Foo k Bar x1 x2 x3 x4))))
        (if (<= k 0) (+ (x4) (x5)) (Bar)))))
  (slide
   #:title
   "Rekursion zum Nachdenken"
   (vc-append
    20
    (code
     (define Foo
       (lambda (k x1 x2 x3 x4 x5)
         (letrec ((Bar
                   (lambda ()
                     (set! k (- k 1))
                     (Foo k Bar x1 x2 x3 x4))))
           (if (<= k 0) (+ (x4) (x5)) (Bar))))))
    (item "Aufruf von Foo:")
    (code
     (Foo
      4
      (lambda () 1)
      (lambda () -1)
      (lambda () -1)
      (lambda () 1)
      (lambda () 0)))
    (item
     (number->string
      (Foo
       4
       (lambda () 1)
       (lambda () -1)
       (lambda () -1)
       (lambda () 1)
       (lambda () 0)))))))
;;;;End LISP-Wizard
```

`eval> (require 'LISP-Wizard) ==>` ↪ Abbildung 3.11 S. 446



Legende:
Beispiel: `LISP-Wizard` ↪ S. 444

Abbildung 3.11: Slide LISP-Wizard

3.5 Primär Objekt-Orientierung

Objekt-Orientierung[24] verkörpert
viel mehr als eine Programmierungstechnik.
Sie ist eine Denkwelt der gesamten Softwareentwicklung!

Von der mächtigen LISP-(Denk)Welt wurde bisher ein relevanter Ausschnitt behandelt. Nun stellt sich die Frage, welche der skizzierten Optionen sollte man (im Zweifel) vorrangig in Betracht ziehen, wenn es um die konkrete Entwicklung einer Anwendung geht. Richtschnur ist das Ziel, die Software in einer ausreichenden Qualität, termingerecht, mit möglichst geringem Aufwand zu erstellen. Dabei wird die Qualität von Software bestimmt durch ihre:

1. *Leistungsfähigkeit*
   ≡ Die Software erfüllt die gewünschten Anforderungen.

2. *Zuverlässigkeit*
   ≡ Die Software arbeitet auch bei ungewöhnlichen Bedienungsmaßnahmen und bei Ausfall gewisser Komponenten weiter und liefert aussagekräftige Fehlermeldungen (Robustheit).

[24] Zur umfassenden Bedeutung der Objekt-Orientierung siehe z. B. ↪ [31, 100].

3. *Durchschaubarkeit & Wartbarkeit*
   ≡ Die Software kann auch von anderen Programmierern als dem Autor verstanden, verbessert und auf geänderte Verhältnisse eingestellt werden.

4. *Portabilität & Anpassbarkeit*
   ≡ Die Software kann ohne großen Aufwand an weitere Anforderungen angepasst werden.

5. *Ergonomie*
   ≡ Die Software ist leicht zu handhaben.

6. *Effizienz*
   ≡ Die Software benötigt möglichst wenig Ressourcen.

Aufgrund der langjährigen Fachdiskussion über die *Objekt-Orientierung*[25], wollen wir annehmen, dass dieses Paradigma etwas besseres ist, als das *was die Praktiker immer schon gewusst und gemacht haben*. Damit stellt sich die Kernfrage: Wenn das objekt-orientierte Paradigma die Lösung ist, was ist eigentlich das Problem? Des Pudels Kern ist offensichtlich das Unvermögen, komplexe Systeme mit angemessenem wirtschaftlichen Aufwand zu meistern. Objekt-Orientierung verspricht deshalb (↪ [108]),

- einerseits komplexere Systeme erfolgreich konstruieren und betreiben zu können und
- andererseits die Erstellungs- und Wartungskosten von komplexen Systemen zu senken.

Bewirken soll diesen Fortschritt primär eine wesentliche Steigerung der Durchschaubarkeit der Modelle in den Phasen des Lebenszyklus (↪ Abbildung 1.13 S. 51). Die Grundidee ist:

> Ein „Objekt“ der realen (oder erdachten) Welt bleibt stets erhalten. Es ist über die verschiedenen Abstraktionsebenen leicht verfolgbar. Das gewachsene Verständnis über die Objekte der realen Welt verursacht eine größere Modelltransparenz.

Was ist ein Objekt im Rahmen der Erarbeitung eines objekt-orientierten Modells?

[25] Das Koppelwort *Objekt-Orientierung* ist hier mit Bindestrich geschrieben. Einerseits erleichtert diese Schreibweise die Lesbarkeit, andererseits betont sie Präfix-Alternativen wie z. B. Logik-, Regel- oder Muster-Orientierung.

Exkurs: Begriff *Modell*
Der Begriff „Modell“ ist mit divergierenden Inhalten belegt. In der mathematischen Logik ist „Modell“ das Besondere im Verhältnis zu einem Allgemeinen: Das Modell eines Axiomensystems ist eine konkrete Inkarnation (≈ Interpretation) dieses Systems. In der Informatik wird der Begriff in der Regel im umgekehrten Sinne verwendet: Das Modell ist das Allgemeine gegenüber einem Besonderen.

Entscheidend für klassische objekt-orientierte Ansätze ist die Frage, zu welchem *Zeitpunkt* die Auswertung (hier: der Typen der Argumente) vollzogen wird. Prinzipiell kann es zur Zeit der Compilierung (*statisches Konzept*) oder zur Laufzeit (*dynamisches Konzept*) erfolgen. Java™ unterstützt den Compilierungszeit-Polymorphismus. *Smalltalk* (↪ [73]) ist beispielsweise ein Vertreter des Laufzeit--Polymorphismus.

Nun wird (leider) beinahe in jedem Paradigma der Begriff „Objekt“ verwendet; beispielsweise bei *Data-driven Programming*, *Pattern-driven Programming* oder auch bei *Programming in Logic*. Wir konzentrieren uns hier auf eine Objekt-Orientierung im Sinne des *Klasse-Instanz*-Modells (↪ Abschnitt 2.8 S. 327). Dessen Vor- sowie Nachteile verdeutlichen wir an dem simplen Beispiel, einen Schreibtisch mit 4 Schubladen abzubilden (↪ Abschnitt 3.5.1 S. 449).

Exkurs: *Data-driven Programming*
Die „Daten-gesteuerte Programmierung“[26] (↪ z. B. [1]) ist keine leere Worthülse oder kein Pleonasmus[27], weil offensichtlich Programme von Daten gesteuert werden. Der Programmierer definiert selbst eine *dispatch*-Funktion[28], die den Zugang zu den datentypabhängigen Operationen regelt. Die Idee besteht darin, für jeden anwendungsspezifischen Datentyp ein selbstdefiniertes Symbol zu vergeben. Jedem dieser Datentyp-Symbole ist die jeweilige Operation als ein Attribut zugeordnet.[29] Quasi übernehmen damit die „Datenobjekte“ selbst die Programmablaufsteuerung. Erst zum Zeitpunkt der Auswertung eines Objektes wird die zugeordnete Operation ermittelt.

Exkurs: *Pattern-driven Programming*
Ist die Auswahl der Operation abhängig von mehreren Daten im Sinne eines strukturierten Datentyps, dann liegt es nahe, das Auffinden der Prozedur als einen Vergleich zwischen einem Muster und einem Prüfling mit dem *Ziel: Passt!* zu konzipieren (↪ Abschnitt 2.4.3 S. 249). Ein *allgemeingültiger Interpreter*, der dieses Passen feststellt, wird dann isoliert, d. h. aus dem individuellen Programmteil herausgezogen.

Da eine Datenbeschreibung (z. B. eine aktuelle Menge von Argumenten) prinzipiell mehrere Muster und/oder diese auf mehr als einem Wege entsprechen könnte, ist eine Abarbeitungsfolge vorzugeben. Die *Kontrollstruktur* der Abarbeitung ist gestaltbar im Sinne eines Beweises, d. h. sie geht

[26] engl.: *data-directed programming* oder auch *data-driven programming*

[27] Als Pleonasmus wird eine überflüssige Häufung sinngleicher oder sinnähnlicher Ausdrücke bezeichnet.

[28] engl.: *dispatch* ≡ Abfertigung

[29] LISP ermöglicht diese Attributzuordnung über die Eigenschaftsliste (P-Liste, ↪ Abschnitt 2.2.3 S. 191), die klassische LISP-Systeme für jedes Symbol bereitstellen.

aus von einer Zielthese und weist deren Zutreffen nach, oder im Sinne einer Zielsuche, d. h. sie verfolgt einen Weg zum zunächst unbekannten Ziel. Anhand der Kontrollstruktur kann die Verknüpfung von Daten und Prozedur mittels der Technik des Mustervergleichs[30] als Vorwärtsverkettung[31] und/oder als Rückwärtsverkettung[32] erfolgen.

Exkurs: *Programming in Logic*
Aus der Tradition des muster-gesteuerten Prozeduraufrufs sind Schritt für Schritt leistungsfähige regel- und logik-orientierte Sprachen mit Rückwärtsverkettung[33], wie z. B. *PROLOG*[34], entstanden.

3.5.1 `class%` versus *A-Liste*

Wir nehmen als simples Beispiel an, es sei ein Schreibtisch (`Desk`) mit vier Schubladen (`Drawers`) abzubilden. Im imperativ-geprägten Paradigma (↪ S. 449) geht es nun um die passenden Variablen, deren Werte Schritt für Schritt berechnet (modifiziert) werden. Im objekt-orientierten Paradigma (↪ S. 451), hier in der Form des *Klasse-Instanz*-Modells (↪ Abschnitt 2.8 S. 327) geht es um das Senden von Nachrichten an Objekte (Instanzen einer Klasse) damit entsprechende Methoden (Funktionen) ausgeführt werden.

Die vier Schubladen haben alle die gleiche Breite, Länge und Höhe, die wiederum abhängig sind von der Breite, Länge und Höhe des Schreibtisches. Im imperativ-geprägten Fall bilden wir die Eigenschaften des Schreibtisches als Assoziationsliste (*A-Liste*, ↪ Abschnitt 2.2.2 S. 183) ab. Im objekt-orientierten Fall als `class`-Konstrukt `Desk%`. Der „Behältertyp“ für die vier Schubladen ist jeweils ein `vector`-Konstrukt (↪ Abschnitt 2.3 S. 215). Das Ziehen einer Schublade (`pull!`) und das wieder Zurückschieben (`push!`) wird bezogen auf die jeweilige Schublade gezählt.

Imperativ-geprägte Konstruktion

```
eval>  ;;Für den Mutator set-cdr!
       (require rnrs/base-6)
       (require rnrs/mutable-pairs-6)

eval> (define Desk
        (lambda (w l h d)
          `((Width ,w)
            (Length ,l)
            (Height ,h)
            (Drawers ,d))))

eval> (define Drawer
          (lambda (w l h)
```

[30] engl.: *pattern matching*
[31] engl.: *forward chaining*
[32] engl.: *backward chaining*
[33] engl.: *backward chaining rule languages*
[34] *PROgramming in LOGic* ↪ [11, 40]

```
         '((Width ,w)
           (Length ,l)
           (Height ,h)
           (Drawn ,0))))

eval> ;;Für die Selektoren
      (define Mein-assq
       (lambda (Key A_Liste)
        (cond ((null? A_Liste) null)
              ((eq? Key (caar A_Liste))
               (car A_Liste))
              (#t (Mein-assq Key (cdr A_Liste))))))

eval> (define get-Drawer
        (lambda (n)
          (vector-ref
           (car (cdr (Mein-assq 'Drawers MyDesk)))
          n)))

eval> (define get-Drawn (lambda (D)
        (car (cdr (Mein-assq 'Drawn D)))))

eval> (define pull!
       (lambda (D)
        (set-cdr! (Mein-assq 'Drawn D)
         (list
          (+ 1
            (car (cdr (Mein-assq 'Drawn D))))))))

eval> (define push!
       (lambda (D)
        (set-cdr! (Mein-assq 'Drawn D)
         (list
          (+ 1
            (car (cdr (Mein-assq 'Drawn D))))))))

eval>  ;;Breite, Länge, Höhe und Anzahl
      (define W 50.0)
eval> (define L 70.0)
eval> (define H 80.0)
eval> (define D 4)

eval> ;;Eine Schublade
     (define MyDrawer
        (Drawer (/ W D) (/ L D) (/ H D)))

eval>  ;;Vier gleiche Schubladen
      (define Drawers
        (make-vector D MyDrawer))

eval>  ;;Der Schreibtisch
      (define MyDesk (Desk W L H Drawers))
```

Bemerkenswert ist, dass für das Erhöhen des Zählers `Drawn` die gewählte Datenstruktur recht aufwendig ist. Die notwendigen Selektoren

sind relativ mühsam zu durchschauen. Wenn auch auf die andernen Eigenschaften, wie beispielsweise die Höhe einer Schublade, zugreifbar gemacht würden, wären weitere, wenig durchschaubare Selektoren notwendig.

Exemplarisch bedienen wir im Folgenden die erste Schublade:

```
eval> (define D1 (get-Drawer 0))
eval> (pull! D1)
eval> (push! D1)
evalA (get-Drawn D1) ==> 2
```

Objekt-orientierte Konstruktion

Der Schreibtisch und die Schublade sind jeweils eine eigene Klasse, wobei die Klasse `Drawer%` innerhalb der Klasse `Desk%` definiert ist. Der Modul `Desk` exportiert den Schreibtisch `MyDesk` mit den gleichen Maßen, wie bei der imperativ-geprägten Lösung (↪ S. 449).

```
;;;;Simples OO-Beispiel:
;;;; Schreibtisch mit 4 Schubladen
;;;; Version 1.0
(module Desk scheme
  (provide MyDesk)
  (define Desk%
    (class object%
      (init w l h d)
      (define
       Drawer%
       (class object%
         (init w l h)
         (define Width w)
         (define Length l)
         (define Height h)
         (define Drawn 0)
         (define/public
          get-Width
          (lambda () Width))
         (define/public
          get-Length
          (lambda () Length))
         (define/public
          get-Height
          (lambda () Height))
         (define/public
          get-Drawn
          (lambda () Drawn))
         (define/public
          push!
          (lambda () (set! Drawn (+ Drawn 1))))
         (define/public
          pull!
          (lambda () (set! Drawn (+ Drawn 1))))
```

```
        (super-new)))
      (define Width w)
      (define Length l)
      (define Height h)
      (define
       Drawers
       (make-vector
        d
        (new
         Drawer%
         (w (/ w d))
         (l (/ l d))
         (h (/ h d)))))
      (define/public get-Width (lambda () Width))
      (define/public
       get-Length
       (lambda () Length))
      (define/public
       get-Height
       (lambda () Height))
      (define/public
       get-Drawer
       (lambda (n) (vector-ref Drawers n)))
      (super-new)))
  (define MyDesk
    (new Desk% (w 50.0) (l 70.0) (h 80.0) (d 4))))
;;;;End Module Desk
```

```
eval> (require 'Desk)
eval> (send (send MyDesk get-Drawer 0) pull!)
eval> (send (send MyDesk get-Drawer 0) push!)
eval> (send (send MyDesk get-Drawer 0) get-Drawn)
      ==> 2
```

Unstrittig ist, das mit etwas Aufwand die imperativ-geprägte Lösung noch verbessert werden könnte (Hülle um die *A-Liste*). Die „Leichtigkeit", mit der die objekt-orientierte Lösung ihre innere Struktur schützt (verbirgt), wird aber nur ansatzweise erreicht. Unstrittig ist aber auch, das bei einer Erweiterung der Aufgabe die objekt-orientierte Lösung ihre Überlegenheit demonstrieren kann.

Nehmen wir z. B. an, es sollte zusätzlich (später) das Material des Schreibtisches gespeichert werden. Mit Hilfe der Vererbungsoption definieren wir dazu eine Klassse `Wood-Metal-Desk%`. Damit diese Klasse als ihre Oberklasse `Desk%` nutzen kann, muss der Modul `Desk` (↪ S. 451) seine Klasse exportieren. Wie erweitern daher sein `provide`-Konstrukt wie folgt:

```
(module Desk scheme
  (provide Desk%  MyDesk)
   ...)
```

Rudimentär definieren wir die Klasse `Wood-Metal-Desk%` folgendermaßen:

```
eval> (define Wood-Metal-Desk%
        (class Desk%
          (init m)
          (define Material m)
          (super-new)))
```

Damit lässt sich ein entsprechender Schreibtisch definieren:

```
eval> (define MyWoodDesk
        (new Wood-Metal-Desk%
          (m "Oak")
          (w 50.0)
          (l 70.0)
          (h 80.0)
          (d 4)))
eval> (send (send MyWoodDesk get-Drawer 0) pull!)
eval> (send (send MyWoodDesk get-Drawer 0) get-Drawn)
      ==> 1
eval> ;;Der exportierte MyDesk
      ;; ist geschützt.
      (define MyDesk
        (new Wood-Metal-Desk%
         (m "Oak") (w 50.0) (l 70.0)
         (h 80.0) (d 4)))
      ==> ERROR ...
      ;define-values: cannot re-define
      ; a constant: MyDesk
```

3.5.2 Zusammenfassung: Primär Objekt-Orientierung

In der objekt-orientierten (Denk-)Welt (*OO-Paradigma*) „wimmelt“ es von Objekten, die in der hier erörterten Ausprägung als Instanzen von Klassen entstehen und sich gegenseitig Nachrichten senden. Ihre Klassen beschreiben die Eigenschaften; d. h. sie definieren passive Objekteigenschaften (≡ Variablen) und aktivierbare Objekteigenschaften (≡ Methoden). Aufgrund der „LISPischen“ Dualität von Daten und Programm können Zustandsvariablen Klassen als Wert haben und damit aktivierbar sein. Klassen können ihre „Beschreibungen“ erben und vererben.

Im *Klasse-Instanz*-Modell lässt sich ein „Objekt“ der realen (oder erdachten) Welt in ein äquivalentes Objekt im Modell abbilden. Üblicherweise gelangt so die Semantik der Objektbezeichnung in die tiefen Modellschichten und hilft damit, die Durchschaubarkeit (Transparenz) dort zu verbessern.

An der Kommunikation zwischen den Objekten können auch anonyme Objekte und anonyme Klassen beteiligt sein, wie das folgende

„durchdenkenswerte“ Beispiel von geschachtelten Klassen mit Vererbungsbeziehungen zeigt.

Charakteristisches Beispiel für Abschnitt 3.5

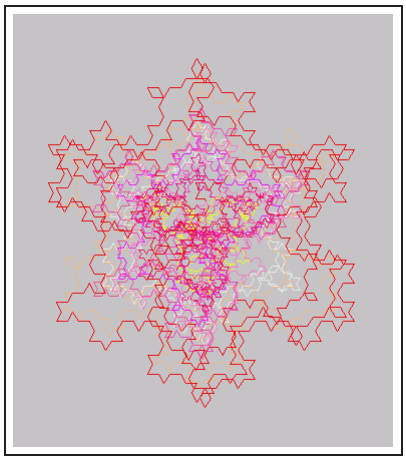
```
;;;;Beispiel für geschachtelte Klassen
;;;; Version 1.0
(module OO scheme
  (provide Level0%)
  (define Level0%
    (class object%
      (define slot "Level0%")
      (define
       LevelI%
       (class Level0%
         (define attribute "LevelI%")
         (define
          LevelII%
          (class LevelI%
            (define
             instance
             (new
              (class object%
                (define foo "Hello World!")
                (define/public
                 get-Foo
                 (lambda () foo))
                (super-new))))
            (define/public
             get-Instance
             (lambda () instance))
            (super-new)))
         (define/public
          get-Attribute
          (lambda () attribute))
         (define/public
          get-LevelII%
          (lambda () LevelII%))
         (super-new)))
      (define/public get-Slot (lambda () slot))
      (define/public
       get-LevelI%
       (lambda () LevelI%))
      (super-new))))
```

Den Modul `OO` nutzen wir, indem wir Nachrichten mit dem `send`-Konstrukt an anonyme Objekte aus seiner exportierten Klasse `Level0%` schikken. Zum Schluss steht als Ergebnis das klassische `"Hello World!"`.

```
eval> (require 'OO)
eval> (send (new (send (new Level0%)
```

```
                        get-LevelI%))
             get-Slot)
       ==> "Level0%"
eval> (send (new (send (new Level0%)
                        get-LevelI%))
             get-Attribute)
      ==> "LevelI%"
eval> (send (send (new (send (new (send
                                    (new Level0%)
                               get-LevelI%))
                        get-LevelII%))
                   get-Instance)
             get-Foo)
       ==> "Hello World!"
```

3.6 Ausblick



Jeder Text von derartiger Länge und „Tiefe“ verlangt ein abschließendes Wort für seinen getreuen Leser. Es wäre nicht fair, nach 455 Seiten, die nächste aufzuschlagen und dann den Anhang zu finden. Daher zum Schluss ein kleiner Ausblick.

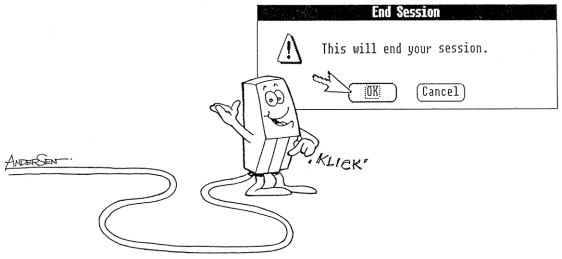
LISP ist und bleibt eine mächtige und offene Programmiersprache mit vielen Konzepten und Möglichkeiten. Sie haben sich einen großen Ausschnitt davon erarbeitet. Buchstabieren mit LISP-Konstrukten fällt Ihnen nicht schwer (`apply begin call/cc define eval false gensym...zero?`). Sie durchschauen das Wurzelwerk von LISP: Die funktionale Wurzel mit ihrem nicht destruktiven Binden (Stichwort: Rekursion) und die imperative Wurzel mit ihrem destruktiven Zuweisen (Stichwort: `set!`). Beide zusammen führen zur objekt-geprägten Konstruktion mit Klassen, Instanzen, Nachrichtenaustausch und vielfältigen

Vererbungsmechanismen. Sie wissen wie Konstruktionsaufgaben LISP-geprägt zu spezifizieren und transparent zu dokumentieren sind.

Was können wir in Zukunft von LISP erwarten? Oder *LISPisch* formuliert: Wie sieht „`call/cc` für LISP“ selbst aus? Unstrittig werden Konzepte und Leistungen aus anderen Programmiersprachen kontinuierlich integriert. Dabei wird eine vermehrte Ausrichtung auf das *mobile Computing* wahrscheinlich bedeutsam sein.

Sicherlich ist LISP mit seinen vielfältigen Dialekten die erste Wahl, wenn es um die Umsetzung von innovativen Ideen auf einer neuen Theoriebasis im Bereich der Softwareentwicklung geht. Zusätzliche Konstrukte, innovative Konstruktionen und nützlichere Konstruktionsempfehlungen können wir erwarten. Vergleiche ich die praktisch erprobten Möglichkeiten Anfang der 90iger Jahre mit den heutigen, so war zumindestens damals ein leistungsfähiger Web-Server in LISP nicht verfügbar. Wir können auf die Entwicklung von LISP daher auch in Zukunft sehr gespannt sein.

Das Buch ist sicherlich (wie viele andere im Fachgebiet Informatik auch) eine Beschreibung, wie der Autor das LISP-Wissen gewonnen hat, das andere schon längst veröffentlicht haben. Trotz alledem wurde versucht, diese Entdeckungsreise aus eigenständiger Sicht zu zeigen. Ein Fachbuch wie dieses ist eigentlich niemals fertig: während man daran arbeitet, lernt man immer gerade genug, um seine Unzulänglichkeiten klarer zu sehen. Schließlich ist es jedoch, inclusive erkannter Unzulänglichkeiten, der Öffentlichkeit zu übergeben. Für diese Unzulänglichkeiten bitte ich Sie abschließend um Ihr Verständnis.



Anhang A

LISP-Systeme

Unterschiedliche Probleme erfordern unterschiedliche Rechnerleistungen, kombiniert mit problemangepassten LISP-Systemen. Ein Allzweckrechner mit einem Allzweck-LISP-System, bestehend aus LISP-Interpreter und/oder LISP-Compiler, stellt stets ein Kompromiss dar, wobei im Einzelfall mehr oder weniger Nachteile in Kauf zu nehmen sind. Für die jeweilige LISP-Hard-/Software-Kombination bestehen zumindest vier Optimierungsaufgaben (↪ z. B. [144]):

1. effizienter Funktionsaufruf
2. Verwalten der Umgebung (*deep access* oder *shallow access*)
3. effizienter Zugriff auf Listenelemente bei ökonomischer Listenrepräsentation
4. Speicherplatzverwaltung (*heap maintenance* mit zweckmäßigem *garbage collection*)

Evaluieren wir z. B.: `(- Foo Bar)`, dann können die Werte von `Foo` und `Bar` vom Typ *Integer* oder *Real* sein, jeweils sind dann andere Operationen durchzuführen. Die Typprüfung zur Laufzeit ist möglichst effizient zu realisieren, da es in LISP-Konstruktionen nur so „wimmelt“ von Funktionen. Beispielsweise nutzten spezielle LISP-Maschinen (↪ Abschnitt A.1 S. 460) dazu Zusatzbits im Speicher (*tagged memories*) und prüften damit hardwaremäßig den Typ. So verfügte der LISP-Rechner *Symbolics 3600* über eine Hardware, die 34 Typen (Symbole, Listen, komplexe Zahlen etc.) unterstützt. Das einzelne Datenwort hat zusätzlich 6 *typetag*-Bits, deren Auswertung gleichzeitig mit der eigentlichen Berechnung geschah.

Die klassische Listenabbildung mit `cons`-Zellen aus zwei Zeigern (↪ Abschnitte 1.2.1 S. 52 und 2.2 S. 160) ist nicht effizient in Bezug auf Speicherplatz und Zugriffszeit. Wenn es gilt, auf viele Listenelemente

zuzugreifen, wenn wir also entlang der Liste „traversieren“, dann ist die Adresse der Zelle, auf die wir als nächste zuzugreifen, in der Zelle enthalten, die gerade im Zugriff ist. Die Nachfolgeradresse kann erst an den Speicher geschickt werden, wenn der vorhergehende Zugriff abgeschlossen ist (*addressing bottleneck*). Zusätzlich verbraucht diese klassische Abbildung viel Speicherplatz, da sowohl der `car`-Zeiger wie der `cdr`-Zeiger den gesamten Adressraum abdecken müssen und dazu entsprechend viele Bits benötigen. Dies ist nicht notwendig, wenn eine Liste kompakter abgebildet wird. Üblich sind sogenannte `vector`-codierte und `structure`-codierte Abbildungen.

Das Konzept der Vektorcodierung geht zunächst von einer linearen Liste aus. Sie enthält keine Sublisten, d. h. keines ihrer Elemente ist wiederum eine Liste. Die einzelnen Elemente sind dann benachbart als Vektorfelder speicherbar. Ist ein Element jedoch wiederum eine Liste, dann wird dies als Ausnahme gekennzeichnet. Jedes Vektorfeld weist durch einen Zusatz (*tag*) aus, ob es ein Element enthält oder eine Adresse, die auf einen neuen Vektor verweist oder unbenutzt ist oder das Ende (`null`) darstellt.

Das Optimierungsproblem liegt in der fixen Länge eines Vektors (↪ Abschnitt 2.3 S. 215). Eine Lösung ist das sogenannte *cdr-coding-representation*-Schema, das LISP-Maschinen, wie z. B. der Rechner *Symbolics 3600*, nutzten. Hierbei verwendet man `cdr`-codierte-Zellen, zusammengesetzt aus einem kleinen `cdr`-Teil und einem relativ großen `car`-Teil. Z. B. ist der `cdr`-Teil 2 Bits und der `car`-Teil 29 Bits. Die vier Möglichkeiten im `cdr`-Teil sind: `cdr`-normal, `cdr`-error, `cdr`-next und `cdr`-nil. `cdr`-next und `cdr`-nil approximieren die Vektorcodierung, `cdr`-normal entspricht der klassischen LISP-Zelle. Eine `cdr`-Codierung mit größerem `cdr`-Teil kann die Seitengröße der Speicherverwaltung des Betriebssystems berücksichtigen (↪ z. B. [49]).

Das Konzept der Strukturcodierung geht davon aus, dass in einer Zelle zusätzlich die Position innerhalb der Liste abgebildet ist. Die Listenstruktur ist als Knoteninformationen eines Baumes gespeichert. Die Listenelemente selbst sind dann als Menge von Tupeln: *Knotennummer-Wert* speicherbar. Das Suchen im Strukturbaum ist effizient gestaltbar, so dass der Zugriff auf Listenelemente im Durchschnitt wesentlich schneller als bei der klassischen Zweizeiger-Zelle ist.

Welche Listenabbildung ist vorzuziehen? Vektorcodierung hat Speicherplatzvorteile, Strukturcodierung hat Zugriffsvorteile. Offensichtlich ist die Wahl davon abhängig, welche Listen und Zugriffe für das jeweilige LISP-Programm typisch sind.

LISP selbst und nicht der Programmierer ist für die Speicherplatzverwaltung verantwortlich, insbesondere für die Bereitstellung (engl.: *allocation*) und Freigabe (engl.: *deallocation*) von `cons`-Zellen im sogenannten *Heap*-Speicher (deutsch: Haufen). Die Freigabe im Sinne einer Speicherbereinigung (engl.: *garbage collection*) ist ein zweistufiges

Verfahren mit *garbage*-Erkennung und *garbage*-Rückgabe. Das Ziel ist es, dass diese *garbage*-Bearbeitung keinesfalls kritische Ausführungen verzögert. Der Zeitpunkt der *garbage*-Bearbeitung und die *Heap*-Teilbereiche, die erfolgversprechend untersucht werden, sind zu optimieren (effiziente „Absterbeordnung“).

Im Laufe der langen LISP-Entwicklung erzielte man wesentliche Verbesserungen bei den skizzierten Optimierungsaspekten: Funktionsaufruf, Umgebungsmanagement, Listenrepräsentation und Speicherplatzmanagement (zur Anfangsgeschichte von LISP ↪ z. B. [175]). So ist *McCarthy's* LISP 1.5, quasi bis Mitte der 60iger Jahre der weltweite LISP-Standard, kaum mit modernen LISP-Systemen, wie z. B. das hier genutzte *PLT Scheme* vergleichbar. Die Ahnenreihe heutiger LISP-Systeme entwickelt aus dem Stammvater LISP 1.5 ist sehr, sehr lang.

In der Praxis stellt sich für jede Programmiersprache die Frage: Gibt es einen möglichst überall akzeptierten Standard? Für LISP lautet die klare Antwort „Jein“ (↪ [148]). Es gab eine Reihe von Bemühungen für LISP einen Standard zu definieren; zu nennen sind z. B. :

1. *EuLISP* ≡ europäischer LISP-Standard, initiiert im Jahr 1985 in Paris und erste Interpreterimplementation ≈ 1990.
   Es ist ein LISP mit der Intention weniger von der Vergangenheit geprägt zu sein (*less encumbered by the past* in Relation zu *Common LISP*). Es sollte nicht so minimalistisch wie Scheme sein und eine starke Ausrichtung auf das objekt-orientierte Paradigma umfassen (`TELOS` — *The EuLisp Object System*)

2. *Common LISP* (↪ [174])
   Common LISP (kurz: CL) ist ein komplexer, vielfältige Paradigmen umfassender LISP-Dialekt, der 1994 als *ANSI/X3.226-1994* Standard definiert wurde. Näheres zum Standard und zur Community z. B.
   ↪ `http://www.lispworks.com/` (Zugriff: 6-Jan-2010).

3. *Scheme* (↪ [151]);
   aktuelle Fassung:
   ↪ `http://www.r6rs.org/final/r6rs.pdf` (Zugriff: 06-Jan-2010))
   Scheme wurde von *Gerald Jay Sussman* und *Guy Lewis Steele Jr.* am MIT für Experimentierzwecke mit Programmierparadigmen entworfen. Scheme verzichtet auf Kompatibilität mit klassischen LISP-Konzepten zugunsten eines mathematisch stringenten Konzeptes.

   R6RS (*Revised*[6] *Report on the Algorithmic Language Scheme*) ist ein moderner LISP-Dialekt, der z. B. in der hier genutzten Form von *PLT-Scheme* (↪ S. 15 und S. 467) eine leistungsfähige Entwicklungsumgebung bereitstellt.

Erst in den 70iger Jahren reiften die heute aktuellen Konzepte, so dass die Standardbemühungen laufend überrollt wurden. Mit *Common LISP* konnten diese Verbesserungen, z. B. die lexikalische Bindungsstrategie oder generische Funktionen, im Standard integriert werden. Allerdings blieb *Common LISP* in wichtigen Punkten hinter Dialekten wie *Scheme* zurück, insbesondere weil man der Kompatibilität mit damals existierenden LISP-Dialekten besondere Priorität einräumte, um die riesige Menge vorhandener Anwendungssoftware leichter auf den Standard umstellen zu können.

Bei den Bemühungen, ein neues (europäisches) LISP zu standardisiern, hatte die Stringenz der Spezifikation vor der Kompatibilität mit vorhandenen LISP-Programmen Priorität. Aus den Erfahrungen mit dem *Common LISP*-Standardisierungsprozess hat man gelernt und spezifiziert den Standard nicht als monolithischen Block eines umfassenden LISP-Systems. Die Spezifikation sieht verschiedene Leistungsschichten vor, z. B. (↪ [177]):

- *Level 0*: Ein *Pure LISP*-System mit dem Umfang von *Scheme*.
- *Level 1*: Ein mittleres LISP-System.
- *Level 2*: Ein großes LISP-System mit dem Umfang von *Common LISP*.

A.1 Rechner für LISP-Systeme

Wir skizzieren hier Rechner für LISP-Systeme ohne eine Bewertung in Bezug auf einzelne Aufgabenklassen (↪ Abschnitt 3.2 S. 403). Auch sind keine Leistungskennwerte angegeben. Dazu sei auf den LISP-spezifischen *Benchmark*-Vorschlag von *Richard P. Gabriel* verwiesen (↪ [67]).

LISP-Systeme sind heute auf allen gebräuchlichen Rechnern verfügbar. Neben LISP-Maschinen (*Symbolic Processors*) kann erfolgreich mit Mainframes, Workstations, Personalcomputern und Supercomputern in LISP Software konstruiert werden. Diese Rechnereinteilung in vier grobe Leistungsklassen benutzen wir, um die Entwicklung und spezielle Eigenschaften zu erörtern. Eine Kurzbeschreibung verbreiteter LISP-Systeme enthält der anschließende Abschnitt A.2 S. 462, in dem auch auf die Rechnerklassen verwiesen wird.

LISP-Maschinen (Geschichte)

Mitte der 70iger Jahre entstand am *MIT* ein Prozessor, speziell ausgerichtet auf die Anforderungen der Symbolverarbeitung. Dieser Rechner wurde unter dem Namen *LISP-Maschine* bekannt. Das MIT vergab an die

Firmen *Symbolics, Inc.* und an *LISP Machine, Inc.* die Lizenz zum Bau und Vertrieb dieses Prozessors. In Konkurrenz zum MIT-Projekt entwickelte man bei *Xerox* (*Palo Alto Research Center*, kurz: PARC) ebenfalls eine kommerziell verfügbare LISP-Maschine.

Im Jahre 1984 stellte *Texas Instruments* den *Explorer* vor, der auf den Arbeiten vom *MIT* (und *PARC*) fußt. Unter dem Begriff *Symbolic Processor* fallen auch die *Inference Machine* und die *Melcom PSI Machine* (Mitsubishi), die erste *PROLOG*-basierte Maschine.

Bei den LISP-Maschinen war z. B. im Jahre 1987 *Symbolics, Inc.* der Marktführer ($\approx$ 2000 Maschinen $\hookrightarrow$ [127]). Der damalige Markterfolg war vornehmlich auf die hervorragende Software-Entwicklungsumgebung zurückzuführen ($\hookrightarrow$ z. B. [26]).

Das *Common LISP*, entwickelt aus dem ursprünglichen *ZetaLISP*, ist durch ein leistungsfähiges *Flavor*-System ergänzt. Mit diesem ist ein dynamisches Windowsystem implementiert. Ein inkrementeller Compiler erleichtert die *edit-compile-debug*-Abläufe. Eine Vielzahl von Werkzeugen aus der Disziplin *Künstliche Intelligenz* war auf diesen LISP-Maschinen lauffähig, z. B. *OPS5* (Verac), *KEE* (IntelliCorp) oder *ART* (Inference Corporation). Hervorzuheben war die ausgezeichnete Grafik-Unterstützung, sowohl schwarz-weiß wie farbig.

Der *Explorer* verfügte über *Common LISP* mit der *Flavor*-Erweiterung der ursprünglichen MIT LISP-Maschine. Ähnlich wie bei der *Symbolics*-Familie waren auch für den *Explorer* der große Arbeitsspeicher (bis 16 Megabyte) und der große virtuelle Adressraum (bis 128 Megabyte) charakteristisch.

Die *Xerox 1100-Familie* stammt nicht direkt von der MIT LISP-Maschine ab. Sie wurde parallel dazu von Xerox entwickelt. Ihr Schwerpunkt lag auf auf dem LISP-Dialekt *InterLISP-D*, eine Software-Entwicklungsumgebung mit damals neuartiger Unterstützung für den Programmierer. Beispielsweise war in InterLISP-D ein *Programmierassistent* integriert, der Schreibfehler erkannte und beim *Debugging* behilflich war. Für die objekt-orientierte Programmierung gab es statt *Flavors Common LOOPS*, eines der Ausgangssysteme für den Standard *CLOS* (*Common LISP Object System* $\hookrightarrow$ Abschnitt 2.8 S. 327).

Warum gibt es keine LISP-Maschinen mehr auf dem Computermarkt? Unter der Überschrift „*Why Did Symbolics Fail?*“ gibt der *Common Lisp*-Entwickler *Daniel L. Weinreb* eine glaubhafte Erklärung.[1] Dort bemerkt er abschließend: „*... I thought I' d never see Lisp again. But now I find myself at ITA Software, where were writing a huge, complex transaction-processing system (a new airline reservation system, initially for Air Canada), whose core is in Common Lisp. We almost certainly have the largest*

[1] *Dan Weinrebs* blog:
$\hookrightarrow$ `http://danweinreb.org/blog/why-did-symbolics-fail` (Zugriff: 22-Jan-2010)

team of Common Lisp programmers in the world. Our development environment is OK, but I really wish I had a Lisp machine again.“.

Mainframes

Sowohl für die IBM-Welt (*LISP/VM*, *Franz LISP*, *Le_LISP* etc.) als auch für die Siemens-Welt sind leistungsfähige LISP-Systeme vorhanden (↪ z. .B. [167]). Gravierende Nachteile gegenüber den LISP-Implementationen auf speziellen Maschinen, Workstations oder PCs liegen in der schwierigen (oder sogar ganz fehlenden) Graphik-Unterstützung. Ihr Vorteil gegenüber diesen dezidierten Maschinen ist die problemlose Integration mit bestehenden Datenbanken und Anwendungspaketen.

Trotz der leistungsfähigen Software-Entwicklungsumgebung, die LISP im Vergleich zu *COBOL* auch auf Mainframes bietet (*Do What I Mean*-Konzept, Programmierassistent, Debugger, etc.), gibt es vergleichsweise sehr wenige Implementationen.

Workstations / Personalcomputer

Für diese Rechner gibt es LISP-Systeme in sehr großer Zahl (↪ Abschnitt A.2 S. 462). Ihr Vorteil gegenüber den dezidierten LISP-Maschinen liegt einerseits in den geringeren Einstiegskosten und andererseits in der einfachen Integration in bestehende Automationen.

Supercomputer

Für Supercomputer waren (sind?) Rechner der Firma *Cray* ein Maßstab. Cray bietet das *Cray LISP* an. In Abweichung von den obigen Ausführungen soll hier doch eine Leistungsangabe erfolgen, weil Supercomputer heute ausschließlich aufgrund ihrer Rechengeschwindigkeit in Betracht kommen. Von dem *Cray LISP* wird berichtet (*Robert J. Douglas*, ↪ [127]), dass es $\approx 5 \ldots 15$ mal schneller ist als LISP als auf einer *Symbolics 3600* (↪ Abschnitt A.1 S. 460).

A.2 Software

In der folgenden tabellarischen Aufzählung ist für den jeweiligen Interpreter und/oder Compiler der Hersteller bzw. der Vertreiber genannt und die Rechnerklasse, für die das System (ursprünglich) konzipiert wurde. Zusätzlich sind Merkmale und Besonderheiten des Systems skizziert. Berücksichtigt sind Systeme mit historischer Bedeutung und mit aktuellen Installationen. Die folgende Kurzübersicht erhebt keinen Anspruch auf vollständige Nennung der Marktführer. Die Systeme sind in alphabetischer Reihenfolge genannt.

- *Allegro Common LISP*
  (↪ http://www.franz.com/products/allegrocl/ (Zugriff: 22-Jan-2010))
  - Rechnerklasse: Workstation, PC
  - Hersteller: Ürsprünglich Coral Software Corp., P.O. Box 307, Cambridge, MA 02142, USA
  - Merkmale: Allegro Common LISP enthält eine Programmierumgebung abgestimmt auf die *Macintosh*-Welt. Das System ist ein inkrementeller Compiler.
- *BYSO LISP*
  (↪ http://www.jstor.org/pss/3680147 (Zugriff: 22-Jan-2010))
  - Rechnerklasse: PC
  - Hersteller: Levien Instrument Co., Sitlington Hill, P.O. Box 31, McDowell, VA 24458, USA.
  - Merkmale: BYSO LISP ist an Common LISP ausgerichtet, hat jedoch starke Wurzeln im *MacLISP* und im klassischen LISP 1.5. Der Interpreter hat daher dynamische Bindungsstrategie. Der Compiler hat sowohl lexikalische als auch dynamische Bindung. Das *Closure*-Konstrukt ermöglicht eine objekt-geprägte Programmierung.
- *Cambridge LISP*
  (↪ http://www.atarimagazines.com/startv1n4/cambridgelisp.html (Zugriff: 22-Jan-2010))
  - Rechnerklasse: PC
  - Bezugsquelle: Philgerma GmbH, Barerstraße 32, D-8000 München 2.
  - Merkmale: Cambridge LISP basiert auf *PSL* (Portable Standard LISP).
- *Emacs LISP*
  (↪ http://www.gnu.org/software/emacs/manual/elisp.html (Zugriff: 22-Jan-2010))
  - Rechnerklasse: PC
  - Hersteller (Copyright): Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.
  - Merkmale: GNU Emacs Lisp ist Grundlage und Komponente des sehr verbreiteten Editors Emacs. Es ist primär beeinflusst von *Maclisp* und etwas von *Common Lisp*.
- *Exper LISP*
  - Rechnerklasse: PC
  - Hersteller: ExperTelligence, Inc., 559 San Ysidro Road, Santa Barbara, CA 93108, USA.
  - Merkmale: Exper LISP enthält eine Graphik, die unter Exper LOGO bekannt wurde. Diese Graphik ermöglicht 3D-Abbildungen. Neben ExperLISP gibt es vom gleichen Hersteller einen *Common LISP* Compiler.
- *Franz LISP*
  (↪ http://www.franz.com/ (Zugriff: 22-Jan-2010))
  - Rechnerklassen: Mainframe und Workstation

  - Hersteller: Franz, Inc. 1141 Harbor Bay Parkway, Alameda, CA 94501, USA.
  - Merkmale: Franz LISP ist ein sehr weit verbreiteter, klassischer LISP-Dialekt mit starker *Common LISP* Ähnlichkeit. Während *Common LISP* primär lexikalische Bindung hat, bindet Franz LISP normalerweise dynamisch.

- *Golden Common LISP (GC LISP)*
  (↪ `http://www.goldhill-inc.com/common.htm` (Zugriff: 22-Jan-2010))
  - Rechnerklasse: PC
  - Hersteller: Gold Hill Computers, Inc. 163 Harvard St., Cambridge, MA 02139, USA.
  - Merkmale: GC LISP 286/386 ist ein Common LISP für die PC-Welt. Mit *HALO* steht eine leistungsfähige Graphik-Toolbox zur Verfügung. Aufgrund des *Common LISP*-Umfanges ist *GC LISP* wesentlich größer und zum Teil nicht so schnell, wie z. B. *muLISP*.

- *Guile*
  (↪ `http://www.gnu.org/software/guile/guile.html` (Zugriff: 22-Jan-2010))
  - Rechnerklasse: PC
  - Hersteller (Copyright): Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.
  - Merkmale: GNU Guile ist ein Interpreter für Scheme.

- *IBUKI Common LISP* (↪ *Kyoto Common LISP*)
  - Rechnerklasse: Workstation
  - Hersteller: Ibuki Inc., 1447 N. Shoreline Blvd. Mountain View, CA 94043, USA
  - Merkmale: Ibuki umfasst keinen vollständigen Compiler, sondern ein Preprozessor für C Compiler. Der LISP-Code wird in C übersetzt.

- *InterLISP-D*
  (↪ `http://en.wikipedia.org/wiki/Interlisp` (Zugriff: 22-Jan-2010))
  - Rechnerklasse: LISP-Maschine (Xerox)
  - Hersteller: Xerox, Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94304, USA
  - Merkmale: ↪ Abschnitt A.1 S. 460

- *IQ-LISP / IQ-CLISP*
  - Rechnerklasse: PC
  - Hersteller: Integral Quality, Inc. P.O. Box 31970, Seattle, WA 98103-0070, USA.
  - Merkmale: IQ-LISP ist ähnlich zu UCI LISP, einem Dialekt von MacLISP. IQ-LISP unterstützt den 8087 Arithmetikprozessor. IQ-CLISP ist die Common LISP Version. Es verfügt über ein einfaches Window-System.

- *Kyoto Common LISP*
  (↪ `http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/lisp/impl/kcl/0.html` (Zugriff: 22-Jan-2010))

  - Rechnerklasse: Workstation
  - Hersteller: Kyoto Common LISP (KCL) wurde am *Research Institute for Mathematical Sciences, Kyoto University, Japan* entwickelt.
  - Merkmale: KCL ist ein vollständiges *Common LISP*. Es hat bei geladenem Interpreter und Compiler einen Bedarf von 1.4 Megabyte Arbeitsspeicher. Der Kern von KCL ist in C geschrieben. KCL hat einen zweistufigen Compiler. Zunächst wird nach C umgesetzt und dann von C in den compilierten Code.

- *Le_LISP*
  (↪ http://www.softwarepreservation.org/projects/LISP/le_lisp/doc/lelisp15.25-refman-en.pdf (Zugriff: 22-Jan-2010))
  - Rechnerklassen: Mainframe (IBM unter VM/UTS), Workstation, PC
  - Hersteller: Le_LISP wurde im Europäischen KI-Zentrum *INRIA*, Frankreich entwickelt. Die Portierung auf PCs erfolgte durch Act Informatique.
  - Merkmale: Eine große Untermenge von *Common LISP* ist abgedeckt. Das Konzept der virtuellen Maschine erleichtert eine Portierung zwischen den Rechnerklassen. Le_LISP enthält ein Paket-Konzept, Stringverarbeitung, Vektoren und Konstrukte für nicht-lokale Kontrollstrukturen.

- *Lucid Common LISP*
  (↪ http://www.lispworks.com/products/lcl.html (Zugriff: 22-Jan-2010))
  - Rechnerklasse: Workstation
  - Hersteller: Lucid, Inc. 707 Laurel Street, Menlo Park, CA 94025, USA.
  - Merkmale: Lucid Common LISP ist eine Entwicklungsumgebung für den professionellen Produktionseinsatz. Neben dem vollen *Common LISP* Sprachumfang sind EMACS-Stil-Editor, Window-System und ein Flavor-System vorhanden.

- *muLISP*
  (↪ http://de.wikipedia.org/wiki/MuLISP (Zugriff: 22-Jan-2010))
  - Rechnerklasse: PC Hersteller: Soft Warehouse, Inc. POB 11174, Honolulu, Hawaii 96828; Microsoft, Inc. 10700 Northrup Way, Bellevue, WA 98004, USA.
  - Merkmale: muLISP deckt ca. 400 der ca. 800 *Common LISP* Konstrukte ab. MuLISP hat den Ruf besonders schnell zu sein und wenig Speicherplatz zu benötigen. Aufgrund der Pseudocode-Compilation ist der Speicherbedarf relativ gering.

- *Nil's LISP*
  - Rechnerklasse: PC
  - Hersteller: A. Fittschen, Weusthoffstraße 6, D-2100 Hamburg 90.
  - Merkmale: Nil's LISP basiert primär auf TLC-LISP und enthält Komponenten von UCI LISP. Ein Flavor-System ist verfügbar. Nil's LISP wurde an der Universität Hamburg eingesetzt.

- *PSL*
  (↪ http://en.wikipedia.org/wiki/Portable_Standard_Lisp (Zugriff: 22-Jan-2010))

  - Rechnerklasse: Workstation
  - Hersteller: University of Utah, Computer Science Department, Salt Lake City, UT 84112, USA.
  - Merkmale: Versuch eines LISP-Standards vor dem *Common LISP* Standard. Starker Einfluss von FranzLISP und MacLISP.

- *PC Scheme*
  (↪ `http://www.cs.indiana.edu/scheme-repository/imp.html` (Zugriff: 22-Jan-2010))
  - Rechnerklasse: PCs: MS-DOS Rechner
  - Hersteller: Texas Instruments, Dallas, USA.
  - Merkmale: Ein gestrafftes LISP mit lexikalischer Bindungsstrategie. Unterstützt wird die objekt-orientierte Programmierung durch *SCOOPS*. Die Portierung von *Common LISP Programmen* ist nicht unproblematisch.

- *Power LISP*
  (↪ `http://www.powerlisp.com/` (Zugriff: 22-Jan-2010))
  - Rechnerklasse: PC
  - Hersteller: MicroProducts, Inc. 370W. Camino Gardens Boulevard, Boca Raton, Florida 33432, USA.
  - Merkmale: Vollständige Implemetation von Interlisp (↪ [182]). Enthält CLISP (*"conversational LISP"), das laut Hersteller eine LISP-Syntax ermöglicht, die mehr an konventionelle Programmiersprachen angepasst ist.*

- *SoftWave LISP*
  (↪ `http://www.gnu.org/software/gcl/` (Zugriff: 22-Jan-2010))
  - Rechnerklasse: PC
  - Hersteller: SoftWave, Seattle, WA 98103-1607, USA
  - Merkmale: Einfacher LISP-Interpreter, der sich am LISP 1.5 orientiert. Nur für Schulungszwecke geeignet.

- *T*
  (↪ `http://www.paulgraham.com/thist.html` (Zugriff: 22-Jan-2010))
  - Rechnerklasse: Workstation
  - Hersteller: `T` Project, Yale Department of Computer Science, Box 2158, New Haven, CT 06520, USA
  - Merkmale: `T` ist eine Scheme Implementation, wobei Scheme eine Untermenge von `T` bildet. `T` hat objekt-geprägte Konstrukte und Makros eingebaut, verfügt über einen optimierten Compiler und unterstützt insbesondere die Fehlersuche (↪ [168]).

- *TLC-LISP* (↪ [2])
  - Rechnerklasse: PC
  - Hersteller: The LISP Company, Redwood Estates, CA 95044, USA

  - Merkmale: TLC-LISP zeichnet sich durch eine klare Grundkonzeption aus, die die Realisierung von „*first class*“-Objekten zum Ziel hat. Es weicht daher in einigen Punkten von *Common LISP* ab. Versucht wird das *ZetaLISP* für PCs verfügbar zu machen. Integriert sind daher ein Klassensystem und ein Paket-Konzept. Zusätzlich enthält TLC-LISP die von LOGO bekannte *Turtle Grafik*. Der Lieferumfang umfasst auch den Compiler und Interpreter in LISP-Quellecode.

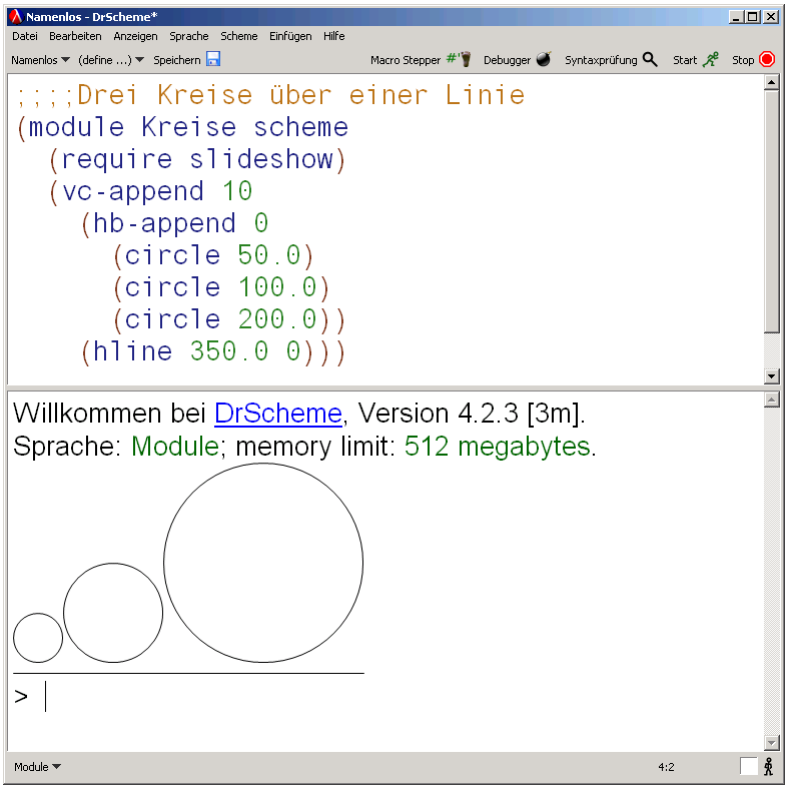
- *UCI LISP*
  (↪ `http://openlibrary.org/b/OL4149344M/new_UCI_LISP_manual` (Zugriff: 22-Jan-2010))
  - Rechnerklasse: PC (ursprünglich für PDP-10 entwickelt)
  - Bezugsquelle: H. Peters, Eielkampsweg 50, D-2083 Halstenbeck.
  - Merkmale: Eine Implementation für PCs in vollem Umfang der ursprünglichen PDP-10-Version — nur noch historisch bedeutsam.
- *XLISP*
  (↪ `http://www.xlisp.org/` (Zugriff: 22-Jan-2010))
  - Rechnerklasse: PC
  - Hersteller: David Betz, Public Domain Programm
  - Merkmale: XLISP integriert Konzepte von *Scheme* und *Common LISP* (↪ z. B. [114]). Es enthält Konstrukte für die objekt-orientierte Programmierung. Eine Programmierumgebung fehlt.
- *ZetaLISP*
  (↪ `http://www.emacswiki.org/emacs/ZetaLisp` (Zugriff: 22-Jan-2010))
  - Rechnerklasse: LISP-Maschine
  - Hersteller: Hersteller der jeweiligen LISP Maschine, Grundlage bildet die MIT LISP Maschine (↪ Abschnitt A.1 S. 460).
  - Merkmale: Flavor-Konzept (↪ Abschnitt 2.8 S. 327)

A.3 PLT-Scheme Übersicht

A.3.1 Systemarchitektur

PLT-Scheme ist eine mächtige Entwicklungsumgebung. Mit *DrScheme* lassen sich Programme in unterschiedlichen (experimentellen) Sprachen entwickeln. Die Abbildung A.1 S. 468 zeigt die hier überwiegend genutzten Fenster. Für die Fehlersuche ist ein leistungsfähiges Diagnosewerkzeug (*Debugger*) verfügbar (↪ *Toolbar*). Zur Analyse von Makros gibt es einen *Macro stepper*, der von der *Toolbar* aktiviert wird. Ein Beispiel zeigt Abbildung A.2 S. 469.

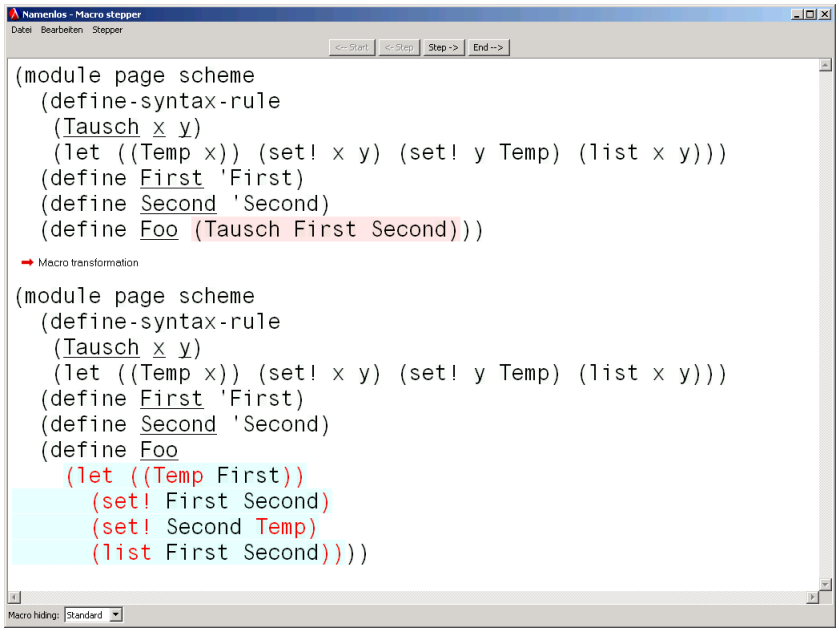
Mitte der 90iger Jahre gründete *Matthias Felleisen* PLT mit dem Ziel, Material mit einer didaktisch fundierten Grundlage, insbesondere eine einfach handhabbare Entwicklungsumgebung, für „*Newcomer*“ in der Programmierung bereitzustellen. Mit *Matthew Flatt* kam der Einfluss von



Legende:

Oberer Rand	≡ Toolbar
Obere Fenster	≡ Definitionen
Unteres Fenster	≡ Interaktionen (Ein- & Ausgaben)
Weitere Fenster	hier ausgeblendet; z. B. Modulbrowser

Abbildung A.1: PLT-Scheme (DrScheme) — Übersicht —



Legende:
Macro stepper aktivierbar aus der *Toolbar* (↪ Abildung A.1 S. 468).

Abbildung A.2: PLT-Scheme (DrScheme) — Macro stepper —

MrEd (grafischer Editor) und mit *Robert B. Findler*, *Shriram Krishnamurthi* sowie *Cormac Flanagan* der von *DrScheme* hinzu.[2]

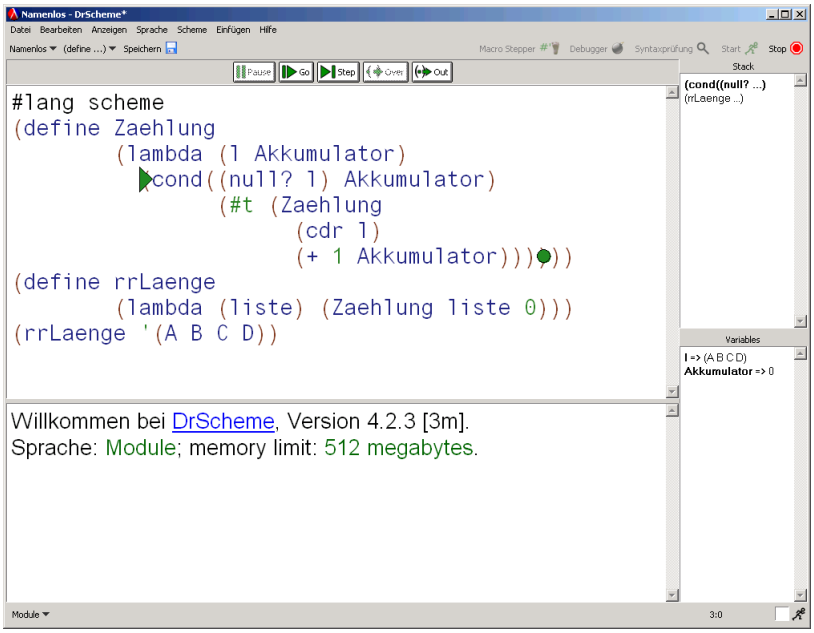
A.3.2 Ausgewählte Scheme-Konstrukte

Darstellung der Konstrukte anhand von Beispielen in der verkürzten Notation; hier statt:
`eval>` < *sexpr* >`==>`< *sexpr* >`;Kommentar` (↪ Abschnitt 1.1.2 S. 20)
kurz:
< *sexpr* >`==>`< *sexpr* >`;Kommentar`

Konstruktoren

```
(cons 'Programmieren '(ist schwierig))
(list (* 2 3) 7 (+ 4 4) 9) ==> (6 7 8 9)
```

[2] Näheres zur PLT-Historie und PLT-Entwicklung ↪ `http://en.wikipedia.org/wiki/PLT_Scheme` (Zugriff: 1-Feb-2010).



Legende:
↪ Scheme-Quellcode S. 116.

Abbildung A.3: Beispiel: *PLT-Scheme Debugger* — Funktion: `rr-Laenge`

```
(append '(A B) '(C D)) ==> (A B C D)
(define Foo (+ 2 3))  ;bindet Foo an den Wert 5
(lambda (x y) (- x y)) ==> #<procedure> ;anonyme Funktion
((lambda (x y) (- x y)) 3 2) ==> 1 ;Funktionsapplikation
(let ((x 3) (y 2)) (- x y)) ==> 1  ;lokale Variablen
(let* ((x 2) (y (* 3 x))) (+ x y)) ;lokale Variablen
                              ==> 8  ; sequentiell gebunden.
;;Rekursiver Bezug auf lokale Variable Foo
(letrec ((Foo (lambda (L)
                (cond((null? L) 0)
                      (#t (+ 1 (Foo (cdr L))))))))
        (Foo '(A B))) ==> 2
(define Karl (lambda (x y) (* (- x 4) y)))
(define (Karl x y) (* (- x 4) y)) ;kürzere Schreibweise
(Karl 2 3) ==> -6

; Falls z.B. module compat
(putprop 'Marx 'Doku "Berühmter Mann")
(getprop 'Marx 'Doku) ==> "Berühmter Mann"

(define-struct Konto
   (Nummer
   Inhaber
   (Stand #:auto))
   #:transparent
   #:mutable
   #:auto-value 0)
(define S1
   (make-Konto
     1
     'Mustermann))
S1 ==> #(struct:Konto 1 Mustermann 0)

(define Konto%
  (class object%
    (init id name)
    (field (Nummer id))
    (define Inhaber name)
    (field (Stand 0))
    (super-new)
    (define/public get-Inhaber
      (lambda () Inhaber))
    (define/public set-Inhaber!
      (lambda (neuerInhaber)
        (set! Inhaber neuerInhaber)))))
(define I1 (new Konto%
                (id 1)
                (name 'Mustermann)))
I1 ==> #(struct:object:Konto% ...)
```

Selektoren und Mutatoren

```
(define Motto '(LISP macht Freu(n)de))
(define Mein-Symbol Motto)
(car Motto) ==> LISP
(cddr Motto) ==> (Freu (n) de)

;;Vorab (require rnrs/base-6) und
;; (require rnrs/mutable-pairs-6)
(set-car! Motto 'Lernen)
(set-cdr! Motto '(ist mühsam))
Motto ==> (Lernen ist mühsam)
Mein-Symbol
  ==> (Lernen ist mühsam);Nebeneffekt!

(Konto-Inhaber S1) ==> Mustermann
(set-Konto-Inhaber! S1 'SAP)

(send I1 set-Inhaber! 'SAP)
(send I1 get-Inhaber) ==> SAP
```

Prädikate und Fallunterscheidung

```
(define Foo 'A)
(define Baz Foo)
(eq? Foo Baz) ==> #t ;benennen dasselbe "Objekt"
(eq? (cons 'A 'B) (cons 'A 'B))
  ==> #f ;verschiedene cons-Zellen
(equal? (cons 'A 'B) (cons 'A 'B))
  ==> #t prüft Wertegleichheit
(zero? (- 2 2)) ==> #t ;prüft numerisch 0
(null? '()) ==> #t ;prüft auf leere Liste
(not (+ 2 3)) ==> #t ;Alles ungleich #f ist wahr
(number? (+ 3 4)) ==> #t
(member 'B '(A B C)) ==> (B C);benutzt equal?
(memq (list 'B) '(A (B) C)) ==> #f ;benutzt eq?
(procedure? (lambda() 'OK))
  ==> #t ;prüft ob Funktion
<  <=   =  >  >=  even?  odd?; für Zahlen

(define Anzahl (lambda (L)
  (cond((null? L) 0)                ;1. Klausel
       (#t (+ 1 (Anzahl (cdr L))))));2. Klausel
  )))
(Anzahl '(Er Sie Es)) ==> 3

(and 1 2 3) ==> 3 ; "true", da ungleich #f
(and null #f (/ 1 0)) ==> #f ;Abbruch
                        ;nach null-Erkennung
```

```
(or #f 2 (/ 1 0)) ==> 2 ;Abbruch bei 2 weil #t

(Konto? S1) ==> #t
(is-a? I1 Konto%) ==> #t
```

Sequenz und Iteration

```
(define Foo
  (begin '(A B)
         '(- 3 4)
         (+ 2 3)))
Foo ==> 5 ;begin gibt letzten Wert zurück

(define Foo
  (begin0 '(A B)
          '(- 3 4)
          (+ 2 3)))
Foo ==> (A B) ;begin0 gibt ersten Wert zurück

(map <funktion> <liste>) ==> ;wendet die Funktion
 ; auf jedes Element der Liste an
 ; und gibt die Ergebnisse als Liste aus.
(map (lambda (x) (+ 2 x)) '(1 2 3)) ==> (3 4 5)
(do ((<variable_1> <anfangswert> <step>) ... )
    (<praedikat>
      <sexpr_1> ... <sexpr_n>) ;Abbruchklausel
    <body>) ==> <value-sexpr_n>
(do ((i 1 (+ i 1)))
    ((> i 5) i)
    (display " ") (display i))
==> 1 2 3 4 56
```

Input-/Output-Interface

```
(define Meine-Ausgabe
  (open-output-file
    "D:\\bonin\\temp\\Schrott.txt"))
Meine-Ausgabe ==>
  #<output-port:D:\bonin\temp\Schrott.txt>
(write 'Dies-ist-Sexpr-1 Meine-Ausgabe)
(newline Meine-Ausgabe)
(write 'Dies-ist-Sexpr-2 Meine-Ausgabe)
(close-output-port Meine-Ausgabe)

(print "Alles klar?") ==> "Alles klar?"
(display "Alles klar?") ==> Alles klar?
(printf "Alles ~a klar \n ~s"
        "doch"
```

```
        "oder nicht?") ==>
Alles doch klar
 "oder nicht?"
;\n  newline
;~a  displays next argument
;~s  writes the next argument

;;Vorab (require scheme/pretty)
(pretty-print '(lambda
   ( x ) (
          + x 7
            )))
==> (lambda (x) (+ x 7))

(display
  ; 60 = Anzahl der Splaten
  (pretty-format '(define ...) 60))
==> Eingerückte Ausgabe

(define Meine-Eingabe
  (open-input-file
    "D:\\bonin\\temp\\Schrott.txt"))
(read Meine-Eingabe) ==>
Dies-ist-Sexpr-1
(read Meine-Eingabe) ==>
Dies-ist-Sexpr-2
(read Meine-Eingabe) ==> #<eof>
(close-input-port Meine-Eingabe)

(call-with-output-file
  "D:\\bonin\\temp\\Schrott.txt"
    #:exists 'append
    (lambda (out)
      (display "Hello Lisp World!" out)
      (newline out)))

(port? (current-input-port)) ==> #t

;Vorab (require net/url)
(define Foo (string->url "http://www.hegb.de"))
(read-line (get-pure-port Foo))
(close-input-port Foo)

;Vorab (require scheme/gui/base)
(define frame
  (new frame% (label "Hello World!")
              (width 300)
              (height 300)))
(define msg
```

```
  (new message% (parent frame)
     (label "LISP macht Freu(n)de")))
(new button% (parent frame)
  (label "Bitte klicken!")
 (callback (lambda (button event)
  (send msg set-label "Button geklickt"))))
  ==> #(struct:object:button% ...)
(send frame show #t)
```

Grafik

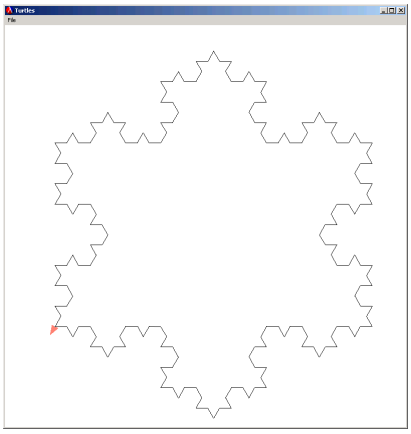
```
; Vorab (require scheme/gui/base)
(define Frame
  (new frame% (label "Ein rotes Rechteck")
              (width 400)
              (height 300)))
(define Canvas
  (new canvas% (parent Frame)))
(define Device-Context
  (send Canvas get-dc))
(define Red-Pen
  (make-object pen% "RED" 10 'solid))
(define Figure
  (lambda (dc)
    (send dc set-pen Red-Pen)
    (send dc draw-rectangle 5 120 380 60)))
(send Frame show #t)
(sleep/yield 1)
(Figure Device-Context)
```

Turtle-Grafik

In LISP ist, wie auch ursprünglich in LOGO, die *Turtle*-Grafik ($\approx$ Zeichenstift für gerade Linien); genannt nach einem Roboter, der ähnlich einer Schildkröte aussah, verfügbar.[3] Kern der Turtel-Grafik sind die drei Befehle: `move`, `draw` und `turn`. In *PLT-Scheme* gibt es zwei *Turtle*-Grafikarten:

1. Traditionelle imperativ-geprägte *Turtle*-Operationen, die in einem festen Fenster stattfinden.
   `(require graphics/turtles)`
2. Funktionale *Turtle*-Operationen, die ein Turtle-Bild als Argument und als Funktionswert haben.
   `(require graphics/value-turtles)`

[3] Mathematiker und Psychologe *Seymour Papert*, $\star$ 1-Mar-1928 in Pretoria, Südafrika, entwickelte diesen Roboter, um Kindern das Programmieren anschaulich zu machen.



Legende:
Module `Koch-Kurve` ↪ S. 477; hier $n = 3$:

```
eval> (Koch-Kurve 3 1500)
```

Abbildung A.4: Turtle-Grafik: Fraktalfigur „Schneeflocke“

Mit der „klassischen Turtle“ ist der Modul `Koch-Kurve` (↪ S. 477) programmiert (Ergebnis ↪ Abbildung A.4 S. 476). Mit einer solchen Figur hat sich schon um 1900 der schwedische Mathematiker *Helge von Koch*, ⋆ 25-Jan-1870 und † 11-Mar-1924 in Stockholm, befasst. Sie wird ihm zu Ehren daher auch als tiradische Koch-Insel oder als Koch'sche Schneeflocke bezeichnet.

Exkurs: *Fraktale Objekte* (*Fractals*)
Der Begriff *Fractals* stammt vom polnisch-französischen Mathematiker *Benoit B. Mandelbrot* (↪ [117]). Man bezeichnet damit Objekte, die sich nicht mehr mit den Begriffen wie Linie oder Fläche beschreiben lassen. Solchen Objekten zwischen Linie (*Dimension* = 1) und Fläche (*Dimension* = 2) ordnet man eine nicht ganzzahlige, eine *fraktale* (gebrochene) Dimension zwischen 1 und 2 zu. Aufgrund wissenschaftlicher Vorarbeiten des französischen Mathematikers *Gaston Julia* spricht man in diesem Zusammenhang auch von *Julia*-Mengen.

Kennzeichned für *Fractals* sind die beiden Eigenschaften:

- Selbstähnlichkeit,
  d. h. man findet die Form des Gesamten in jedem Detail wieder.
- Verkrumpelung, d. h. es treten keine glatten Begrenzungen auf, so dass eine Länge oder ein Flächeninhalt nicht zu bestimmen ist.

Beim genauen Hinschauen findet man solche *Fractals* überall in der Natur (↪ [118]). Der Blumenkohl-Kopf ist ein populäres Beispiel. Ein praxisnahes Software-Konstruktionsgebiet ist die Computer-Animation. Man modelliert, z. B. mit stochastischen Fraktalen, detailreiche Gebirgsketten. Die

Koch'sche Schneefocke gehört (wie auch das populäre „Apfelmännchen") zur *deterministischen* (kausal abhängigen) *Fraktale*, da sie mit einer einfachen Rekursionsformel beschreibbar ist. (Fraktale Kunstwerke ↪ z. B. [139]).

```
;;;;Fraktale: Koch-Kurve
;;;; mit imperativer Turtle-Grafik
(module Koch-Kurve scheme
  (require graphics/turtles)
  (provide Koch-Kurve)
  (define Koch-Kurve
    (lambda (n l)
      (let* ((startpunkt
              (lambda ()
                (turn 180)
                (move 300)
                (turn 90)
                (move 200)
                (turn 90)))
             (fractale
              (lambda (n l)
                (cond
                 ((= n 0) (draw l))
                 (#t
                  (fractale (- n 1) (/ l 4))
                  (turn 300)
                  (fractale (- n 1) (/ l 4))
                  (turn 120)
                  (fractale (- n 1) (/ l 4))
                  (turn 300)
                  (fractale (- n 1) (/ l 4)))))))
        (startpunkt)
        (fractale n l)
        (turn 120)
        (fractale n l)
        (turn 120)
        (fractale n l))))
  (turtles))
```

Die bekannte Kinderzeichnung: „*Das Haus des Nikolaus*" (↪ Abbildung A.5 S. 478) verdeutlicht die funktions-geprägten *Turtle*-Konstrukte. Dabei ist das Haus ohne Absetzen des Stiftes unter der Bedingung, dass keine Linie doppelt gezeichnet wird, zu malen.

```
;;;;Kinderzeichnung Nikolaus-Haus
;;;; mit funktionaler Turtle-Grafik
(module Nikolaus-Haus scheme
 (require graphics/value-turtles)
 (provide Nikolaus-Haus)
```

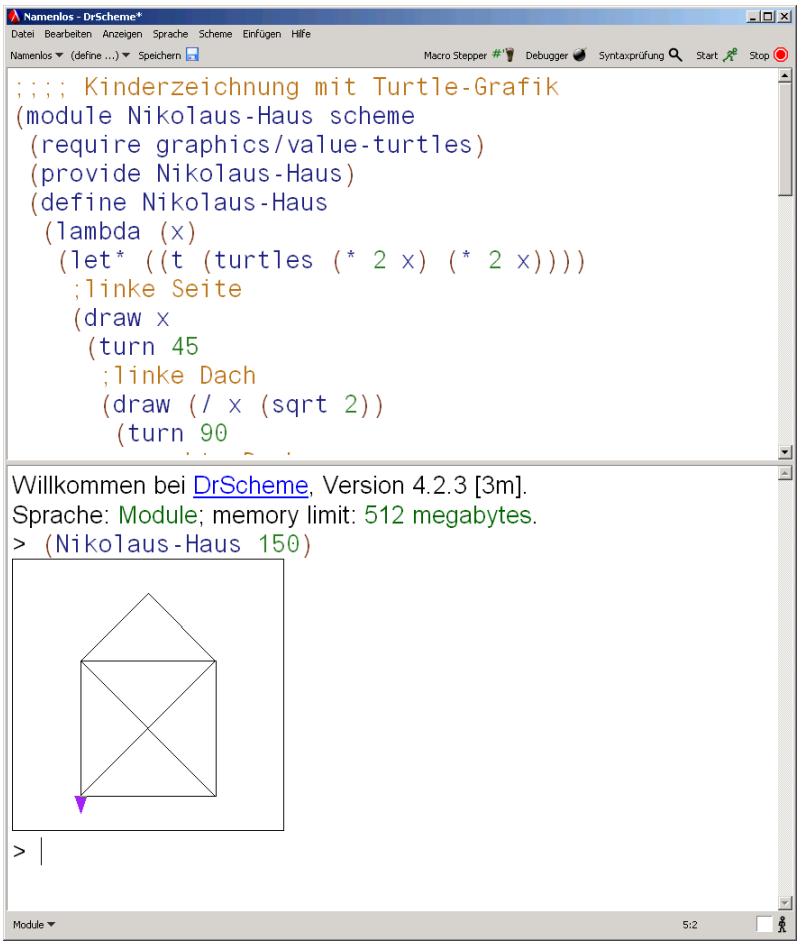
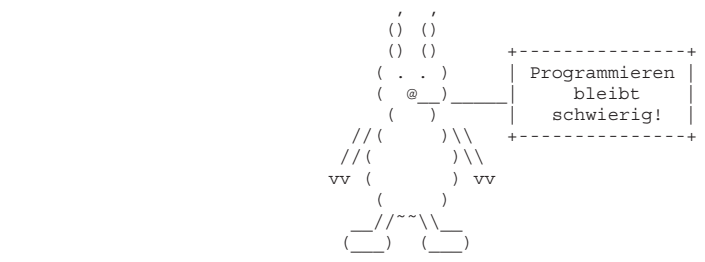


Abbildung A.5: Turtle-Grafik: „Das Haus des Nikolaus“

```
(define Nikolaus-Haus
 (lambda (x)
  (let* ((t (turtles (* 2 x) (* 2 x))))
   ;linke Seite
   (draw x
    (turn 45
     ;linke Dach
     (draw (/ x (sqrt 2))
      (turn 90
       ;rechte Dach
       (draw (/ x (sqrt 2))
        (turn 90
         ;Schräglinie
         (draw (* x (sqrt 2))
          (turn 225
           ;Grundline
           (draw x
            (turn 225
             ; zum Ausgangspunkt
             (draw (* x (sqrt 2))
              (turn 135
               ; Dachbodenlinie
               (draw x
                (turn 90
                 ; rechte Seite
                 (draw x
                  ;Ausgangspunkt
                  (turn 180
                   (move (* (/ x 4 ) 3)
                    (turn 270
                     (move (/ x 2) t)
                     ))))))))))))))))))))
```

Zum Schluss wurde mit den beiden *Turtle*-Grafiken nochmals die Prägung einer Konstruktion durch das jeweilige Paradigma demonstriert — hier: *imperativ-geprägt* versus *funktional-geprägt*.



Tabellenverzeichnis

Abbildungsverzeichnis

Anhang B

Literatur

Notationshinweise:

- Literaturangaben zum Vertiefen des Stoffes dieses Manuskripts sind vor grauem Hintergrund ausgewiesen.
- In der Literatur wird LISP nicht immer in Großbuchstaben geschrieben, obwohl es ein Akronym für *List Processing* ist. Die jeweilige Schreibweise ist hier beibehalten.

Literaturverzeichnis

[1] Harold Abelson / Gerald Jay Sussman / Julie Sussman; Structure and Interpretation of Computer Programs, Cambridge, Massachusetts u. a. (The MIT Press/McGraw-Hill) 1985.

[2] John Allen; Anatomy of LISP, New York u. a. (McGraw-Hill) 1978.

[3] John R. Allen; The Death of Creativity, in: AI Expert, San Francisco 2(1987)2, pp. 48 – 61.

[4] John R. Anderson / Robert Farrell / Ron Sauers; Learning to Program in LISP, in: Cognitive Science, Norwood 8 (1984), pp. 87 – 129.

[5] John R. Anderson / Albert T. Corbett / Brian J. Reiser; Essential LISP, Reading, Massachusetts u. a. (Addison-Wesley) 1987.

[6] American National Standard / Institute of Electrical and Electronics Engineers; Guide to Software Requirements Specifications, Standard 830, July 20, 1984.

[7] John Backus; Can Programming Be Liberated from the von Neumann Style? A Functional Style and It's Algebra of Programs, Communications of the ACM, Vol. 21, No. 8, August 1978, pp. 613 – 641.

[8] Avron Barr / Edward A. Feigenbaum (Eds.); The Handbook of Artificial Intelligence, Volume II, Chapter VI: Programming Languages for AI, (HeurisTech Press/William Kaufmann) 1982.

[9] Clemens Beckstein; Integration objekt-orientierter Sprachmittel zur Wissensrepräsentation in LISP, Institut für Mathematische Maschinen und Datenverarbeitung, Friedrich- Alexander-Universität Erlagen-Nürnberg, Diplomarbeit im Fach Informatik 1985.

[10] F. Beichter / O. Buchegger / O. Herzog / N. E. Fuchs; SLAN-4, A Software Specification and Design Language, in: Floyd/Kopetz, 1981, S. 91 – 108.

[11] Fevzi Belli; Einführung in die logische Programmierung mit Prolog, Mannheim Wien Zürich (Bibliographisches Institut), 2. Auflage 1988.

[12] Fevzi Belli / Hinrich Bonin; Qualitätsvorgaben im Hinblick auf Softwarefehler, in: M. Bartsch / D. Hildebrand (Hrsg.); Der EDV-Sachverständige, Workshop der GRVI, Schmitten 1986, Stuttgart (B. G. Teubner) 1987, S. 172 – 198.

[13] Fevzi Belli / Hinrich Bonin; Probleme bei der Hinterlegung wissensbasierter Systeme bezüglich ihrer Dokumentation — Erstüberlegungen und Thesen -, in: D. J. Hildebrand / T. Hoene (Hrsg.); Software — Hinterlegung in der Insolvenz, Workshop der DGIR, Stuttgart 1988, Stuttgart (B. G. Teubner) 1989, S. 29 – 51.

[14] Hinrich E. G. Bonin; Software-Konstruktion mit LISP, Berlin New York (Walter de Gruyter) 1991, ISBN 3-11-011786-X.

[15] G. Booch; Object-oriented Analysis and Design with Applications, 2nd ed., Redwood City (Benjamin/Cummings), 1994. Deutsche Ausgabe: Objektorientierte Analyse und Design, Mit praktischen Anwendungsbeispielen, Bonn (Addision-Wesley), 1994.

[16] Edmund C. Berkeley / Daniel G. Bobrow (Editors); The Programming Language LISP: Its Operation and Applications, (The M. I. T. Press) 1966, First printing March 1964.

[17] Gerhard Bitsch; Wie man in LISP programmiert — Eine Einführung anhand von PC-Scheme 3.0, CHIP Wissen, Würzburg (Vogel Buchverlag) 1989.

[18] Daniel G. Bobrow; Requirements for Advanced Programming Systems for List Processing, Communications of ACM, Vol. 15, Number 7, July 1972, pp 618 – 627.

[19] Daniel G. Bobrow / Kenneth Kahn / Gregor Kiczales / Larry Masinter / Mark Stefik / Frank Zdybel; Common Loops: Merging Lisp and Object-Oriented Programming, in: ACM Proceedings OOPSLA'86, SIGPLAN Notices, 21 (11) 1986, p. 17 – 29.

[20] Daniel G. Bobrow / David Moon u.a.; Common Lisp Object System Specification, ANSI X3J13 Document 88-002R, American National Standards Institute, Washington, DC, Juni 1988 (veröffentlicht in: SIGPLAN Notices, Band 23, Special Issus, September 1988).

[21] Barry W. Boehm; Software Engineering Economics, New Jersey 1981.

[22] Hinrich Bonin; Die Planung komplexer Vorhaben der Verwaltungsautomation, Heidelberg (R.v.Decker & C.F. Müller) 1988 (Band 3 der Schriftenreihe *Verwaltungsinformatik*).

[23] Hinrich Bonin; Objekt-orientierte Programmierung mit LISP, Arbeitsbericht 1988/7 des Fachbereichs 2 (Systemanalyse) der Hochschule Bremerhaven (ISSN 0176-8158)

[24] Hinrich Bonin; Objektorientierte Programmierung mit LISP, in: Handbuch der Modernen Datenverarbeitung (HMD), 26. Jahrgang, Heft 145, Januar 1989, S. 45 – 56.

[25] Gerhard Brewka / Franco di Primo / Eckehard Groß; BABYLON — Referenzhandbuch V1.1/1, Institut für Angewandte Informationstechnik, Gesellschaft für Mathematik und Datenverarbeitung mbH, Bonn, 1987.

[26] Hank Bromley / Richard Lamson; LISP LORE: A Guide to Programming the LISP Machine, Second Edition, Boston, Dordrecht, Lancaster (Kluwer Academic Publishers) 1987.

[27] Rodney A. Brooks / Richard P. Gabriel / Guy L. Steele Jr.; S-1 Common Lisp Implementation, ACM LISP Conference New York, 1982, pp. 108 – 113.

[28] Rodney A. Brooks / Richard P. Gabriel; A Critique of Common Lisp, Conference record of ACM symposium on LISP and functional Programming: Proceedings Salt Lake City, Utha 1984, pp. 1 – 8.

[29] Rodney A. Brooks; LISP — Programmieren in Common Lisp, München Wien (R. Oldenbourg Verlag) 1987 — Original 1985 "Programming in Common Lisp" (John Wiley).

[30] Fred Brooks; The Mythical Man-Month, (Addison-Wesley) 1975, ISBN 0201006502.

[31] Manfred Broy / Johannes Siedersleben; Objektoientierte Programmierung und Softwareentwicklung — Eine kritische Einschätzung, in: Informatik-Spektrum, Band 25, Heft 1, Februar 2002, S. 3–11.

[32] William H. Burge; Recursive Programming Techniques, Reading, Massachusetts u.a. (Addison-Wesley) 1975.

[33] R. M. Burstall / D. B. MacQueen / D. T. Sannells; HOPE: An Experimental Applicative Language, LISP Conference, Stanford/Redwood Est 1980, pp. 136 – 143.

[34] Firma Calculemus; Comments on TLC-LISP/86 Version 1.51, July 1986.

[35] James Allen Carte; Filtering Typographic Noise from Technical Literature, in: Technical Communication, Third Quarter, Vol. 20, pp. 12 – 13, 1973 (auch erschienen in: Harkins/Plung (Eds.); A Guide for Writing Better Technical Papers, New York (IEEE Press selected reprint series) 1981, pp. 154 – 155.

[36] Eugene Charniak / Christopher K. Riesbeck / Drew V. McDermott / James R. Meehan; Artificial Intelligence Programming, Second Edition, Hillsdale, New Jersey, Hove and London (Lawrence Erlbaum Associates) 1987.

[37] Douglas W. Clark / C. Cordell Green; An Empirical Study of List Structure in Lisp, Communications of ACM, Vol. 20, No. 2, February 1977, pp. 78 – 87.

[38] Volker Claus; Total unstrukturierte Programme, in: Der GMD-Spiegel 4/87 (Gesellschaft für Mathematik und Datenverarbeitung mbH, St. Augustin), S. 46 –50.

[39] William Clinger; Semantics of Scheme, in: Byte, Febraury 1988, pp. 221 –227.

[40] W. F. Clocksin / C. S. Mellish; Programming in Prolog, Berlin New York u.a. (Springer-Verlag) Third Edition, 1987.

[41] Common LISP Object System, 1988 ↪ [20].

[42] Alonzo Church; The Calculi of LAMBDA-Conversion; Princeton University Press, New Jersey, 1941.

[43] Wolfgang Coy / Frieder Nake / Jörg-Martin Pflüger / Arno Rolf / Jürgen Seetzen / Dirk Siefkes / Reinhard Stransfeld (Hrsg.); Sichtweisen der Informatik, Braunschweig, 1992. [Eine gesellschaftspolitisch geprägte Analyse und Perspektive der Informatik.]

[44] H. B. Curry / R. Feys; Combinatory Logic, Vol. I, Studies in Logic and the Foundations of Mathematics, Amsterdam (North-Holland) 1958.

[45] I. Danicic; Lisp Programming, Oxford London Edinburgh u.a. (Blackwell Scientific Publications), reprinted with corrections 1984.

[46] J. Darlington / P. Henderson, / D. A. Turner (Eds.); Functional programming and its applications — An advanced course, Cambridge, London u.a. (Cambridge University Press), auch als UMI Out-of-Print Books on Demand 1989.

[47] Ernst Denert; Software-Modularisierung, in: Informatik-Spektrum 2, 1979, S. 204 – 218.

[48] F. L. De Remer / H. Kron; Programming-in-the-Large versus Programming-in-the-Small, in: Progammiersprachen, 4. Fachtagung der GI, Erlangen, Informatik-Fachberichte 1, (Springer Verlag) 1976, pp. 80 – 89.

[49] L.P. Deutsch; Experience with Microprogrammed Interlisp System, in: Proc. 11th Ann. Microprogramming Workshop, Nov. 1978, pp. 128 – 129.

[50] Edsger W. Dijkstra; Go To Considered Harmful, in: Communications of the ACM, Volume 11 / Number 3 / March 1968, pp. 147 –148.

[51] Deutsches Institut für Normung e. V.; Entscheidungstabelle — Beschreibungsmittel, Berlin, Köln (Beuth Verlag).

[52] Deutsches Institut für Normung e. V.; Sinnbilder nach Nassi-Shneiderman und ihre Anwendung in Programmablaufplänen, Berlin, Köln (Beuth Verlag)

[53] Franco di Primo / Thomas Christaller; A Poor Man's Flavor System, Institut Dalle Molle ISSCO, Universite de Geneve, Working Paper No 47, May 1983.

[54] Domain/Common LISP User's Guide — A Concise Reference Manual, Order No. 008791 (Product developed by Lucid, Inc. of Menlo Park, California) 1986.

[55] Susan Eisenbach (editor); Functional Programming: Languages, Tools and Architectures, Chichester (Ellis Horwood Limited) 1987.

[56] Michael Eisenberg; Programming in Scheme, Cambridge, Massachusetts, London, England (MIT Press) paperback 1990 (1988 by The Scientific Press).

[57] Bleike Eggers / Zimmermann; LISP, Manuskript der Vorlesung WS 1981/82 und WS 1983/84, Technische Universität Berlin, Fachbereich Informatik.

[58] David W. Embley / Barry D. Kurtz / Scott N. Woodfield; Object-Oriented Systems Analysis – A Model-Driven Approach, Englewood Cliffs, New Jersey (Yourdon Press), 1992.

[59] Rüdiger Esser / Elisabeth Feldmar; LISP — Fallbeispiele mit Anwendungen in der künstlichen Intelligenz, Paul Schmitz (Hrsg.): Reihe *Künstliche Intelligenz*, Braunschweig Wiesbaden (Vieweg & Sohn) 1989.

[60] Matthias Felleisen / Robert B. Findler / Matthew Flatt / Shiriram Krishnamurthi; How to design programs — an introduction to programming and computing —, Cambridge, MA (The MIT Press) 2001.

[61] Matthias Felleisen / Daniel P. Friedman; A Closer Look at Export and Import Statements, Computer Languages, Vol. 11, No. 1, 1986, pp. 29 – 37.

[62] Flavor Interface to SCHeme (FISCH), Konzeption und Konstruktion einer LISP-Umgebung für die objektgeprägte Modellsicht der Softwareentwicklung, Projektbericht des Projektes FLAVOR, Fachbereich 2, Hochschule Bremerhaven 1988.

[63] Christiane Floyd / Hermann Kopetz (Hrsg.); Software Engineering — Entwurf und Spezifikation, Stuttgart (B. G. Teubner) 1981.

[64] John K. Foderaro / Keith L. Sklower / Kevin Layer; The FRANZ LISP Manual, June 1983 (University of California).

[65] Charles L. Forgy; OPS5 User's Manual, Department of Computer Science, Carnegie-Mellon University Pittsburg, Pennsylvania 15213, 1981.

[66] Daniel P. Friedman; The Little LISPer, ISBN: 0-574-19165-8, Science Research Associates, 1974.

[67] Richard P. Gabriel; Performance and Evaluation of Lisp Systems, Cambridge, Massachusetts (The MIT Press), 1985 (Third printing 1986).

[68] Richard P. Gabriel; The Why of Y, in: LISP Pointers, Volume 2 Number 2 October-November-December 1988, pp. 15 – 25.

[69] Richard P. Gabriel / Larry M. Masinter; Performance of Lisp Systems, ACM LISP Conference New York, 1982, pp. 123 – 142.

[70] Richard P. Gabriel / Kent M. Pitman; Endpaper: Technical Issues of Separation in Function Cells and Vaule Cells, in: LISP and Symbolic Computation, Vol. 1, Number 1, June 1988, pp. 81 – 101.

[71] Learning LISP, Engelwood Cliffs, New Jersey (Prentice-Hall) 1984 (Handbuch der Firma Gnosis zu LISP auf Apple II).

[72] Martin L. Griss / Anthony C. Hearn; A Portable LISP Compiler, Software Practice and Experience, Vol. 11, 1981, pp. 541 – 605.

[73] Adele Goldberg / Dave Robson; Smalltalk-80: the language, Reading, Massachusetts u. a. (Addison-Wesley) 1983.

[74] Ron Goldman / Richard P. Gabriel; Qlisp: Experience and New Directions, in: ACM/SIGPLAN PPEALS (Parallel Programming: Experience with Applications, Languages and Systems) Volume 23, Number 9, September 1988, pp. 111 – 123.

[75] M. Gordon / R. Milner / C. Wadsworth / G. Cosineau / G. Huet / L. Paulson: The ML-Handbook, Version 5.1, Paris (INRIA), 1984.

[76] Claudio Gutierrez; PROLOG Compared with LISP, ACM LISP Conference New York, 1982, pp. 143 – 149.

[77] John Guttag; Notes on Type Abstraction (Version 2), IEEE Transactions on Software Engineering, Vol. SE-6, No. 1, January 1980, pp. 13 – 23.

[78] Tony Hasemer; A Beginner's Guide to Lisp, Workingham, Bershire u.a.(Addison-Wesley), 1984.

[79] Patrick J. Hall; How to Solve it in LISP, Wilmslow, England (Sigma Press) 1989.

[80] Robert H. Halstead, Jr.; Implementation of Multilisp: Lisp on a Multiprocessor, Conference record of ACM Symposium on LISP and Functional Programming: Proceedings Salt Lake City, Utha 1984, pp. 9 – 17.

[81] Robert H. Halstead, Jr.; New Ideas in Parallel Lisp: Language Design, Implementation, and Programming Tools, in: Ito/Halstead, 1989, pp. 2 – 57.

[82] Christian-Michael Hamann; Einführung in das Programmieren in LISP, Berlin New York (Walter de Gruyter) 1982.

[83] Jürgen Handke; Sprachverarbeitung mit LISP und PROLOG auf dem PC, Band 27: Programmieren von Mikrocomputern, Braunschweig, Wiesbaden (Vieweg & Sohn) 1987.

[84] Williams Ludwell Harrison III; The Interprocedural Analysis and Automatic Parallelization of Scheme Programs, in: LISP and Symbolic Computation, Volume 2, Number 3/4, October 1989, pp. 179 – 396.

[85] Friedrich Haugg / Stefan Omlor; LISP auf PC's — Methoden und Techniken der Symbolischen Datenverarbeitung, München Wien (Carl Hanser Verlag) 1987.

[86] Anthony C. Hearn; Standard LISP, in: ACM SIGPLAN Notices, Vol. 4, No. 9, 1969 (auch Technical Report AIM-90, Artificial Intelligence Project, Stanford University).

[87] Peter Henderson; FUNCTIONAL Programming — Application and Implementation, Englewood Cliffs, New Jersey (Prentice/Hall International) 1980.

[88] Wade L. Hennessey; COMMON LISP, New York, St.Louis San Francisco u.a. (McGraw-Hill) 1989.

[89] Martin C. Henson; Elements of Functional Languages; Computer Science Texts, Oxford, London u. a. (Blackwell Scientific) 1987.

[90] Wolfgang Hesse / Hans Keutgen / Alfred L. Luft / H. Dieter Rombach; Ein Begriffssystem für die Softwaretechnik, in: Informatik-Spektrum, Heft 7 1984, S. 200 – 223 (Arbeitskreis: *Begriffsbestimmung* der Fachgruppe *Software Engineering* im FB 2 der GI)

[91] C. A. R. Hoare; Recursive Data Structures, International Journal of Computer and Information Sciences, Vol. 4, No. 2, 1975, pp. 105 – 132.

[92] Douglas R. Hofstadter; Metamagikum, in: Spektrum der Wissenschaft, April 1983 (S.14–19), Mai 1983 (S.13–19), Juni 1983 (S. 10 – 13), Juli 1983 (S. 6 – 10); Original in Scientific American 1983.

[93] Frederick Holtz; LISP, the language of artificial intelligence (TAB Books, Inc.) 1985.

[94] Ellis Horowitz; Fundamentals of Programming Language, Second Edition, Berlin Heidelberg u.a. (Springer) 1984.

[95] Sheila Hughes; TerminalBuch LISP, München Wien (R. Oldenbourg Verlag) 1987.

[96] Michael Hußmann / Peter Schefe / Andreas Fittschen; Das LISP-Buch, Hamburg (McGraw-Hill) 1988.

[97] Takayasu Ito / Robert H. Halstead, Jr. (Eds.); Parallel Lisp: Languages and Systems, Proceedings US/ Japan Workshop on Parallel LIsp Sendai, Japan, June 5–8, 1989, Lecture Notes in Computer Science 441, Berlin Heidelberg u. a. (Springer-Verlag) 1989.

[98] M. A. Jackson; Grundsätze des Programmentwurfs, 7. Auflage 1986, Darmstadt (Originaltitel: Principles of Program Design; Übersetzung von R. Schaefer und G. Weber)

[99] Ivar Jacobsen / M. Christerson / P. Jonsson / G. Övergaard; Object-Oriented Software Engineering, A Use Case Driver Approach, Workingham (Addision-Wesley) 1992.

[100] Stefan Jähnichen / Stephan Herrmann; Was, bitte, bedeutet Objektorientierung? in: Informatik-Spektrum, Band 25, Heft 4, August 2002, S. 266–275. [Hinweis: Ein Diskussionsbeitrag zu ↪ [31].]

[101] Simon L. Peyton Jones; An Investigation of the Relative Efficiencies of Combinators and Lambda Expressions, LISP Conference New York 1982, pp. 150 – 158.

[102] Michael J. Kaelbling; Programming Languages Should NOT Have Comment Statements, in: SIGPLAN Notices, Vol. 23, No. 10, October 1988, pp. 59 – 60.

[103] Gregor Kiczales / Jim des Rivieres / Daniel G. Bobrow; The Art of the Metaobject Protocol, Cambridge, Massachusetts, London (The MIT Press) 1991.

[104] Sonya E. Keene; Object-Oriented Programming in COMMON LISP: A Programmer's Guide to CLOS, Reading, Massachusetts u.a. (Addison-Wesley) 1989.

[105] Robert M. Keller; Divide an CONCer: Data Structuring in Applicative Multiprocessing Systems, LISP Conference, Stanford/Redwood 1980, pp. 196 – 202.

[106] B. W. Kernighan / P. J. Plauger; The Elements of Programming Style, New York (McGraw-Hill), Second Edition 1978.

[107] Robert R. Kessler; LISP — Objects, and Symbolic Programming, Glenview, Illinois Boston London (Scott, Foresman and Company) 1988.

[108] Won Kim / Frederick H. Lochovsky (Eds.); Object-Oriented Concepts, Databases, and Applications, Reading, Massachusetts (Addison-Wesley) 1989.

[109] Georg Klaus / Heinz Liebscher (Hrsg.); Wörterbuch der Kybernetik, Frankfurt am Main (Fischer Taschenbuch Verlag), Original Berlin (Dietz Verlag) 1967, (4. überarbeitete Fassung 1976).

[110] Dieter Kolb; Interlisp-D, Siemens AG, München, 17.07.85.

[111] H.-J. Kreowski; Algebraische Spezifikation von Softwaresystemen, in: Floyd/Kopetz, 1981, S. 46 – 74.

[112] Robert L. Kruse; Data Structures and Program Design, Englewood Cliffs, New Jersey (Prentice/Hall International) 1984.

[113] P. J. Landin; The mechanical evaluation of expressions, in: The Computer Journal, Vol. 6, No. 4, 1964, pp. 308 – 320.

[114] Johannes Leckebusch; XLisp — Die Programmiersprache der KI-Profis, München (Systhema Verlag) 1988.

[115] D. B. MacQueen / Ravi Sethi; A semantic of types for applicative Languages, ACM LISP Conference New York, 1982, pp. 243 – 252.

[116] Bruce J. MacLennan; Functional Programming, Practice and Theory, Reading, Massachusetts (Addison-Wesley) 1990.

[117] Benoit B. Mandelbrot; Fractals: Form, Chance, and Dimension, San Franzisco (Freeman Press) 1977.

[118] Benoit B. Mandelbrot; The Fractal Geometry of Nature, New York (Freeman Press) 1983.

[119] J. B. Marti / A. C. Hearn / M. L. Griss / C. Griss; Standard Lisp Report, ACM SIGPLAN Notices, Volume 14, Number 10, 1979.

[120] Ian A. Mason; The Semantics of Destructive LISP, Menlo Park (CSLI Lecture Notes Number 5) 1986.

[121] Otto Mayer; Programmieren in Common LISP, Mannheim Wien Zürich (Bibilographisches Institut) 1988.

[122] W. D. Maurer; The Programmer's Introduction to LISP, New York London (American Elsevier), Reissued 1975 (first 1972).

[123] John McCarthy; Recursive Function of Symbolic Expression and Their Computation by Machine, Part I, Comm. ACM 3,4 (April 1960),pp. 184 – 195.

[124] John McCarthy / Paul W. Abrahams / Timothy P. Hart / Michael I. Levin; LISP 1.5 Programmer's Manual, Massachusetts (The M.I.T. Press), Second Edition , Second printing, January 1966.

[125] John McCarthy; LISP-Notes on Its Past and Future, Conference Record of the 1980 LISP Conference.

[126] Donald P. McKay / Stuart C. Shapiro; MULTI — A LISP Based Multiprocessing System, LISP Conference, Stanford / Redwood Est 1980, pp. 29 – 37.

[127] Richard K. Miller; Computers for Artificial Intelligence, A Technology Assessment and Forecast, second edition, SEAI Technical Publications (ISBN 0-89671-081-5) 1987.

[128] M. Minsky; A Framework for Representing Knowledge, in: P. H. Winston (Ed.); The Psychology of Computer Vision, New York (McGraw-Hill) 1975.

[129] Richard Mitchell / Jim McKim; Design by Contract, by Example, Amsterdam (Addison-Wesley Longman), Oktober 2001 (ISBN 978-0201634600).

[130] Mitchell L. Model; Multiprocessing via Intercommunicating LISP Systems, LISP Conference, Stanford / Redwood 1980, pp. 188 – 195.

[131] David Moon; Object-Oriented Programming with Flavors, in: ACM Proceedings OOPSLA'86, SIGPLAN Notices, 21(11) 1986, pp. 1–8.

[132] Joel Moses; The Function of FUNCTION in LISP or WHY the FUNARG Problem Should be Called the Environment Problem, in: SIGSAM Bulletin, New York July 1970, pp. 13 – 27.

[133] Steven S. Muchnick / Uwe F. Pleban; A Semantic Comparison of LISP and SCHEME, LISP Conference, Stanford/Redwood Est 1980, pp. 56 – 64.

[134] Jörg Mühlbacher; Datenstrukturen, München Wien (Carl Hanser) 1975.

[135] Dieter Müller; LISP — Eine elementare Einführung in die Programmierung nichtnumerischer Aufgaben, Mannheim Wien Zürich (Bibliographisches Institut) 1985.

[136] Frieder Nake; Informatik und Maschinisierung von Kopfarbeit, in: [43], S. 181-201. [Zum Schaffen von Frieder Nake ↪ [152].]

[137] A. Narayanan / N. E. Sharkey; An Introduction to LISP, Chichester (Ellis Horwood) 1985.

[138] I. Nassi / B. Shneiderman; Flowchart Techniques for Structured Programming, in: SIGPLAN Notices 8 (1973) 8, p. 12– 26.

[139] H.-O. Peitgen / P. H. Richter; The Beauty of Fractals, Berlin Heidelberg New York (Springer Verlag) 1986.

[140] Rózsa Péter; Rekursive Funktionen in der Komputer-Theorie, Budapest (Akadmiai Kiadó) 1976.

[141] R. Piloty; RTS (Registertransfersprache), Institut für Nachrichtenverarbeitung, Technische Hochschule Darmstadt, 2. Auflage, 8-Jul-1968.

[142] Robert M. Pirsig; Zen and the Art of Motorcycle Maintenance — An Inquiry Into Values, Taschenbuchausgabe: Corgi Books, Deutsche Ausgabe: Zen und die Kunst ein Motorrad zu warten — Ein Versuch über Werte, übersetzt von Rudolf Hermstein, Frankfurt / Main (S.Fischer) 1976.

[143] Kent M. Pitman; Special Forms in LISP, LISP Conference, Stanford / Redwood Est 1980, pp. 179 – 187.

[144] Andrew R. Pleszkun / Matthew J. Thazhuthaveetil; The Architecture of Lisp Machines, Computer, Vol. 20, No. 3, March 1987, pp. 35 – 44.

[145] Vaughan R. Pratt; A Mathematician's View of LISP, BYTE August 1979, pp. 162 – 168.

[146] Mary Ann Quayle / William Weil / Jeffrey Bonar / Alan Lesgolg; The Friendly Dandelion Primer, Tutoring Systems Group University of Pittsburgh 1984.

[147] Christian Queinnec; LISP, übersetzt ins Englisch von Tracy A. Lewis, London (Macmillan Publishers Ltd) 1984 (französische Orginal 1983, Eyrolles Paris).

[148] Christian Queinnec / Jerome Chailloux (Eds.); LISP Evolution and Standardization — Proceedings of the First International Workshop, 22–23 February 1988, Paris, France, Amsterdam (AFCET ISO) 1988.

[149] Rational Software; Unified Modeling Language, Version 1.1, 01-Sep-1997, UML Summary, UML Metamodel, UML Notation Guide, UML Semantics, UML Extension Business Modeling, Object Constraint Specification,
`http://www.rational.com/uml/1.1/`
(Zugriff: 11-Nov-1997)

[150] Jonathan A. Ross; T: A Dialect of Lisp — or, LAMBDA: The Ultimate Software Tool, LISP Conference New York, 1982, pp. 114 – 122.

[151] Jonathan Rees / William Clinger (Editors); Revised 3 Report on the Algorithmic Language Scheme, AI Memo 848a, Artificial Intelligence Laboratory of the Massachusetts Institute of Technology, September 1986.

[152] Karl-Heinz Rödiger (Hrsg.); Algorithmik — Kunst — Semiotik, Hommage für Frieder Nake, Heidelberg (Synchron Wissenschaftsverlag) 2003, ISBN 3-935025-60. [Festschrift für Frieder Nake, einer der großen Pioniere der Computergraphik.]

[153] G. R. Rogers; COBOL-Handbuch, München Wien (R. Oldenbourg Verlag) 1988.

[154] Erich Rome / Thomas Uthmann / Joachim Diederich; KI-Workstations: Überblick — Marktsituation — Entwicklungstrends, Bonn, Reading, Massachusetts u.a. (Addison-Wesley Deutschland) 1988.

[155] J. Rumbaugh / M. Blaha / W. Premerlani / F. Eddy / W. Lorenson; Objekt-oriented Modelling and Design, Englewood Cliffs (Prentice-Hall), 1991

[156] Erik Sandewall; Ideas about Management of LISP Data Bases, Int. Joint Conf. on Artificial Intelligence 1975, pp. 585 – 592.

[157] Erik Sandwall; Programming in an Interactive Environment: the *LISP* Experience, Computing Surveys, Vol. 10, No. 1 March 1978, pp. 35 – 71.

[158] Mina-Jaqueline Schachter-Radig, Entwicklungs- und Ablaufumgebungen für die künstliche Intelligenz — Arbeitsplatzrechner für die Wissensverarbeitung, in: Informationstechnik it, 29. Jahrgang, Heft 5/1987, S. 334 – 349.

[159] Peter Schefe; Informatik — Eine konstruktive Einführung — LISP, PROLOG und andere Konzepte der Programmierung, Mannheim Wien Zürich (Bibliographisches Institut) 1985.

[160] Peter Schefe; Künstliche Intelligenz — Überblick und Grundlagen — Grundlegende Konzepte und Methoden zur Realisierung von Systemen der künstlichen Intelligenz, Mannheim Wien Zürich (Bibliographisches Institut) 1986.

[161] Harald Scheid (Bearbeiter); DUDEN, Rechnen und Mathematik, 4. völlig neu bearbeitete Auflage, Mannheim Wien Zürich (Bibliographisches Institut) 1985.

[162] E. D. Schmitter; Praktische Einführung in LISP — LISP die Sprache der künstlichen Intelligenz, Holzkirchen (Hofacker) 1987.

[163] Peter Schnupp / Christiane Floyd; Software — Programmentwicklung und Projektorganisation, 2. durchgesehene Auflage, Berlin New York (Walter de Gruyter) 1979

[164] Georg Schoffa; Die Programmiersprache Lisp — Eine Einführung in die Sprache der künstlichen Intelligenz, München (Franzis) 1987.

[165] David E. Shaw / William R. Swartout / C. Cordell Green; Inferring LISP Programs from Examples, International joint conference on artificial intelligence: Proceedings. Tbilisi, Georgia 4 (1975), pp. 260 – 267.

[166] Laurent Siklóssy; Let's Talk Lisp, Englewood Cliffs, New Jersey (Prentice-Hall) 1976.

[167] Siemens AG München; INTERLISP — Interaktives Programmiersystem BS2000 Benutzerhandbuch, Bestellnummer: U90015-J-Z17-1, Ausgabe September 1981, Version 4.

[168] Stephen Slade; The T Programming Language: A Dialect of LISP, Englewood Cliffs, New Jersey (Prentice-Hall, Inc.) 1987.

[169] Peter Smith; An Introduction to LISP, A Chartwell-Bratt Student Text, Lund (Studentliteratur) 1988.

[170] Michael Sperber / R. Kent Dybvig / Matthew Flatt / Anton van Straten (Editors); Revised^6 Report on the Algorithmic Language Scheme, 26 September 2007
↪ `http://www.r6rs.org/` (online 3-Dec-2009)

[171] George Springer / Daniel P. Friedman; Scheme and The Art Of Programming, Cambridge, Massachusetts, London, England (MIT Press) 1989.

[172] Guy Lewis Steele Jr.; LAMBDA The Ultimate Declarative, Massachusetts Institute of Technology Artificial Intelligence Laboratory, AI Memo 379, November 1976.

[173] Barbara K. Steele; Strategies for Data Abstraction in LISP, LISP Conference, Stanford/Redwood Est 1980, pp. 173 – 178.

[174] Guy L. Steele Jr. / Scott E. Fahlman / Richard P. Gabriel / David A. Moon / Daniel L. Weinreb; COMMON LISP — The Language, (Digital Press) 1984.

[175] Herbert Stoyan; LISP — Anwendungsgebiete, Grundbegriffe, Geschichte, Berlin (Akademie-Verlag) 1980.

[176] Herbert Stoyan; Programmierstile und ihre Unterstützung durch sogenannte Expertensystem-Werkzeuge, in: Arno Schulz (Hrsg.); Die Zukunft der Informationssysteme — Lehren der 80er Jahre, Berlin Heidelberg New York u.a. (Springer-Verlag) 1986, S. 265 - 275.

[177] Herbert Stoyan / Jerome Chailloux / John Fitch / Tim Krumnack / Eugen Neidl / Julian Padget; Towards a LISP Standard, in: Rundbrief des FA 1.2, Künstliche Intelligenz und Mustererkennung der Gesellschaft für Informatik e.V. Bonn, Nr. 44, Januar 1987.

[178] Herbert Stoyan / Günter Görz; LISP — Eine Einführung in die Programmierung, Berlin Heidelberg New York u.a. (Springer) 1984.

[179] Phillip D. Summers; A Methodology for LISP Program Construction from Examples, in: Journal of Association for Computing Machinery, Vol. 24, No. 1, January 1977, pp. 161 – 175.

[180] Steven L. Tanimoto; The Elements of Artificial Intelligence — An Introduction Using LISP, Rockville, Maryland 1987 (Computer Science Press).

[181] Deborah G. Tatar; A Programmers's Guide to COMMON LISP, Digital Equipment Corporation (Digital Press) 1987.

[182] Warren Teitelman / A. K. Hartley / J. W. Goodwin / D. C. Lewis / D. G. Bobrow / L. M. Masinter; INTERLISP Reference Manual, XEROX Palo Alto Research Center 1974, revised October 1974, revised December 1975.

[183] The Lisp Company, Pob 487 Redwood Estates, CA 95044; TLC-LISP Documentation, Primer, Metaphysics and Reference Manual 1985.

[184] David S. Touretzky; LISP A Gentle Introduction to Symbolic Computation, New York u. a. (Harper & Row) 1984.

[185] David S. Touretzky; The Mathematics of Inheritance Systems, Research Notes in Artificial Intelligence, London (Pitman) 1986.

[186] Christian Wagenknecht; Progammierparadigmen — Eine Einführung auf der Grundlage von Scheme — Stuttgart Leipzig Wiesbaden (B. G. Teubner Verlag) 2004.

[187] Jürgen Wagner; Einführung in die Programmiersprache LISP – Ein Tutorial, Vaterstetten (IWT) 1987.

[188] Daniel Weinreb / David Moon; Lisp Machine Manual, Third Edition March 1981 (Massachusetts Institute of Technology, Cambridge).

[189] Clark Weissman; LISP 1.5 Primer, Boston (PWS Publishers) 1967.

[190] Patrick Henry Winston / Berthold Klaus Paul Horn; LISP — Second Edition, Reading, Massachusetts (Addison- Wesley) 1984 (Deutsche Fassung 1987; Übersetzung von B. Gaertner u. a.).

[191] Niklaus Wirth; Algorithmen und Datenstrukturen, Pascal-Version, 3. überarbeitete Auflage, Stuttgart (B.G. Teubner Verlag) 1983 (1. Auflage 1975).

[192] Jon L. White; LISP: Program Is Data — A Historical Perspective On MACLISP, Proceedings of the MACSYMA. Berkeley, Calif. 1977, pp. 180 – 189.

[193] William G. Wong; TLC-LISP — A fully featured LISP implementation, BYTE , October 1985, pp. 287 – 292.

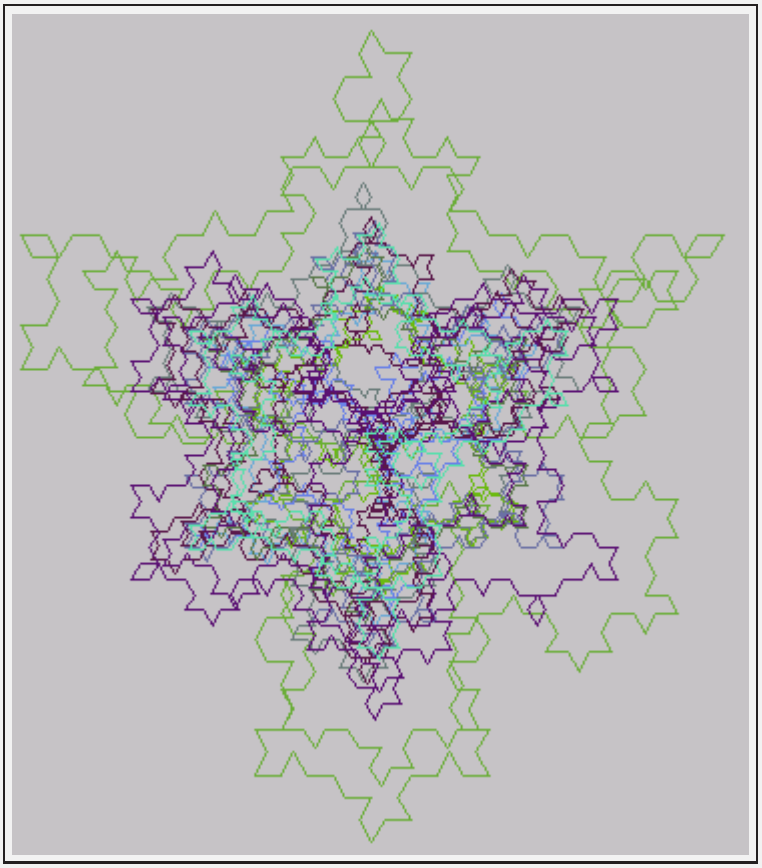
[194] William G. Wong; PC Scheme: A Lexical LISP, MARCH 1987, BYTE, pp. 223 – 226.

[195] Taiichi Yuasa / Masami Hagiya; Introduction to Common Lisp, Boston Orland u.a. (Academic Press, Inc,), orginally published in japanese 1986.

Anhang C

Index

Index



⋆ ⋆ ⋆