Nichnumerisches Programmieren

Helmut Schwichtenberg

Mathematisches Institut der Universität München
Wintersemester 2002/2003

Inhaltsverzeichnis

Vorwort

Das vorliegende Skriptum gibt den Inhalt eines zweiwöchigen Ferienkurses über Nichtnumerisches Programmieren wieder, den ich im Wintersemester 2002/2003 am Mathematischen Institut der Universität München gehalten habe. Es handelt sich um eine vorläufige Ausarbeitung, die an vielen Stellen noch verbesserungs- und ergänzungsbedürftig ist. Als Grundlage dienten neben der Sprachdefinition von SCHEME [**3**] (erhältlich im Internet unter `http://www.swiss.ai.mit.edu/projects/scheme/`) das Buch [**1**] von Abelson und Sussman.

Bedanken möchte ich mich bei Martin Ruckert für seine Hilfe bei der Vorbereitung und Durchführung dieses Kurses. Ferner möchte ich noch die schon etwas länger zurückliegende Hilfe von Holger Benl, Ulrich Berger, Felix Joachimski, Robert Stärk und Michael Stoll anerkennen; von letzterem stammen viele der in diesem Kurs besprochenen Beispiele.

In diesem Kurs habe ich hauptsächlich mit Petite Chez Scheme gearbeitet; es ist kostenlos im Internet erhältlich unter `www.scheme.com`.

München, im Oktober 2002

Helmut Schwichtenberg

KAPITEL 1

Prozeduren

1. Elemente der Programmierung

In diesem Kurs werden wir uns mit der Sprache LISP befassen. Sie wurde von McCARTHY 1960 in einer Arbeit [**4**] mit dem Titel “Recursive Functions of Symbolic Expressions and their Computation by Machine” eingeführt. Der Name LISP steht für LISt Processing. Ursprüngliche Ziele waren etwa das symbolische Differenzieren und Integrieren algebraischer Ausdrücke. LISP ist die zweitälteste Programiersprache, die noch in allgemeiner Benutzung ist; nur FORTRAN ist älter.

In den ersten Implementierungen von LISP wurde kein besonderer Wert auf Effizienz bei numerischen Operationen gelegt, so daß – etwa im Vergleich mit FORTRAN – numerische LISP-Programme langsamer waren. Daraus resultiert ein häufig anzutreffendes Vorurteil über die Ineffizienz von LISP, das aber heute nicht mehr stimmt.

Ein besonderer Vorteil von LISP ist, daß in dieser Sprache Beschreibungen von Prozessen (genannt *Prozeduren*) selbst als Daten bearbeitet werden können. Der traditionelle Unterschied zwischen Daten und Prozeduren verschwindet also. Man kann deshalb besonders leicht Programme schreiben, die andere Programme manipulieren (etwa Interpreter oder Compiler).

Wozu braucht man Programmiersprachen? Zunächst sicher als Hilfsmittel, um einen Rechner zur Lösung von gewissen Aufgaben zu veranlassen. Eine weitere, eher noch wichtigere Aufgabe besteht aber darin, eine Sprache bereitzustellen, in der die Arbeitsweise von Algorithmen klar und eindeutig formuliert werden kann.

Zunächst unterscheiden wir Prozeduren und Daten. In diesem ersten Abschnitt werden wir hauptsächlich numerische Daten behandeln, da sie vertrauter sind.

1.1. Ausdrücke. Um ein Gefühl für das Arbeiten mit LISP zu gewinnen, wollen wir mit einigen einfachen Beispielen beginnen.

```
(+ 3 4) ==> 7
(- 27 19) ==> 8
(* 13 6) ==> 78
(/ 27 9) ==> 3
(/ 10 6) ==> 5/3
(/ 10 6.0) ==> 1.6666666666666667
(+ 2.7 10) ==> 12.7
```

Man beachte, daß wir die sogenannte *Präfixschreibweise* verwenden, in der der Operator immer links geschrieben wird. Dies ist zunächst ungewohnt, hat aber den Vorteil, daß man eine beliebige Anzahl von Argumenten verwenden kann:

```
(+ 3 4 2 10) ==> 19
(* 2 6 3) ==> 36
```

Ein weiterer Vorteil der Präfixschreibweise besteht darin, daß bei mehrfach geschachtelten Ausdrücken wie

```
(* (+ 2
      (* 4 6))
   (+ 3 5 7))
```

eine die Struktur verdeutlichende Schreibweise (*pretty-printing*) möglich ist.

Der Interpreter arbeitet immer in einem gewissen Basiszyklus, nämlich

lesen-auswerten-drucken

(*read-eval-print* loop). Insbesondere ist es nicht nötig, den Interpreter explizit zum Ausdrucken des Wertes aufzufordern.

1.2. Namen und Belegungen. Es ist möglich, Namen Werte zuzuweisen und dann diese Namen als Abkürzungen für die Werte zu benutzen. Am besten versteht man dies anhand eines Beispiels:

```
(define size 2) ==> size
size ==> 2
(* 5 size) ==> 10
(+ (* 5 size) (* size size)) ==> 14
```

Ein weiteres Beispiel:

```
(define pi 3.14159) ==> pi
(define radius 10) ==> radius
(* pi (* radius radius)) ==> 314.159
(define circumference (* 2 pi radius)) ==> circumference
circumference ==> 62.8318
```

1.3. Auswertung von Kombinationen mit primitiven Prozeduren. In zusammengesetzten Ausdrücken werden die Bestandteile von innen nach außen ausgewertet. Man kann sich dies anhand des schon oben betrachteten Ausdrucks

```
(* (+ 2
      (* 4 6))
   (+ 3 5 7))   ==> 390
```

klarmachen.

1.4. Benutzerdefinierte Prozeduren. Außer den primitiven Prozeduren wie + und $*$ kann auch der Benutzer eigene Prozeduren definieren. Parallen zu der Formulierung „das Quadrieren einer Zahl besteht im Multiplizieren der Zahl mit sich selbst“ schreibt man in SCHEME

```
(define (square x) (* x x))
```

Genauer definiert man hierdurch ein Prozedurobjekt (d.h. eine Funktion), und zwar die „Funktion, die jeder Zahl x das Produkt von x mit sich selbst zuordnet“. Dieser sprachliche Ausdruck enthält offenbar x als gebundene Variable. Traditionell verwendet man λ als Bindungsoperator. In SCHEME schreibt man deshalb

```
(define square (lambda (x) (* x x)))
```

Diese Definition von `square` ist gleichwertig mit der obigen.

Zur allgemeinen Form von `lambda`-Ausdrücken siehe die Sprachdefinition [**3**]. Beispiele:

```
(square 3) ==> 9
(square (+ 2 5)) ==> 49
(square (square 3)) ==> 81
```

Wir können `square` benutzen, um daraus weitere Prozeduren zu definieren, etwa

```
(define (sum-of-squares x y) (+ (square x) (square y)))
```

Man erhält

```
(sum-of-squares 3 4) ==> 25
```

1.5. Auswertung benutzerdefinierter Prozeduren. Die Auswertung einer Kombination mit einer definierten Prozedur als Operator geschieht wie folgt.

(1) Die Argumente werden ausgewertet und in einem neu gebildeten Rahmen (frame) an die formalen Parameter gebunden.
(2) Der Kern des `lambda`-Terms wird in diesem neuen Rahmen ausgewertet.

BEMERKUNG. Das Substitutionsmodell für die Auswertung definierter Prozeduren in [**1**] ist irreführend, da es den grundlegenden Begriff des Auswerten eines Terms mit freien Variablen unter einer Belegung vermeidet. Es entspricht auch nicht dem tatsächlichen Arbeiten des Interpreters.

1.6. Bedingte Ausdrücke. Bisher haben wir noch keine Möglichkeit, Fallunterscheidungen auszudrücken, wie etwa in

$$\mathsf{abs}(x) = \begin{cases} x & \text{falls } x > 0, \\ 0 & \text{falls } x = 0, \\ -x & \text{falls } x < 0. \end{cases}$$

In SCHEME gibt es für diesen Zweck die spezielle Form `cond`.

```
(define (abs x)
  (cond ((> x 0) x)
        ((= x 0) 0)
        ((< x 0) (- x))))
```

Eine Alternative ist

```
(define (abs x)
  (cond ((< x 0) (- x))
        (else x)))
```

Statt `cond` kann man auch `if` benutzen.

```
(define (abs x)
  (if (< x 0)
      (- x)
      x))
```

Man beachte, daß `cond` und `if` keine Prozeduren sind, sondern spezielle Formen.

Zur allgemeinen Form von `cond` und `if` siehe die Sprachdefinition [**3**]. Hier spielt offenbar eine wesentliche Rolle, welche Objekte in `test` als wahr oder falsch angesehen werden. In SCHEME gilt die Konvention, daß nur das boolesche Objekt `#f` als falsch gilt, und jedes andere Objekt (insbesondere also auch die leere Liste ()) als wahr gilt. Siehe dazu [**3**], Abschnitt 6.2.

ÜBUNG 1.1. Man beschreibe den Unterschied zwischen `if` und der wie folgt definierten Prozedur `new-if`:

```
(define (new-if test consequent alternative)
  (cond (test consequent)
        (else alternative)))
```

ÜBUNG 1.2. Man schreibe eine Prozedur, die zu k (natürliche Zahl) ein möglichst kleines n (natürliche Zahl) berechnet mit

$$\sum_{m>n} \frac{1}{m!} < \frac{1}{2} \cdot 10^{-k}.$$

Hinweis: Die Summe ist für $n \geq 1$ kleiner als $\frac{1}{n \cdot n!}$.

ÜBUNG 1.3. Die n-te Partialsumme $\sum_{0\le m\le n}\frac{1}{m!}$ der Exponentialreihe ist ein Bruch der Form $\frac{z(n)}{n!}$ mit einer natürlichen Zahl $z(n)$. Schreiben Sie eine Prozedur, die $z(n)$ zu gegebenem n berechnet.

ÜBUNG 1.4. Man schreibe eine Prozedur, die zu jedem k (natürliche Zahl) die natürliche Zahl $a(k)$ berechnet, so daß

$$|e - 10^{-k}\cdot a(k)| < 10^{-k}.$$

Hinweis: Sie werden vermutlich eine Rundungsfunktion `round` benötigen, die als primitive Prozedur bereitgestellt ist.

1.7. Quadratwurzeln nach der Newton-Methode. Als ein weiteres Beispiel behandeln wir eine Implementierung der NEWTON-Methode zur Berechnung von Quadratwurzeln. In der Analysis kann man schon vor der Konstruktion der reellen aus den rationalen Zahlen den folgenden Satz beweisen.

THEOREM 1.5 (Approximation von $\sqrt{a}$). *Es seien $a > 0$ und $x_0 > 0$ gegeben. Die Folge x_n sei rekursiv definiert durch*

$$x_{n+1} := \frac{1}{2}\left(x_n + \frac{a}{x_n}\right).$$

Dann gilt

(1) $(x_n)_{n\in\mathbb{N}}$ ist eine CAUCHYfolge.
(2) Wenn $\lim_{n\to\infty} x_n = b$, so ist $b^2 = a$.

BEWEIS. Wir führen den Beweis (wie in [**2**]) in mehreren Schritten.
1. Durch Induktion über n zeigt man leicht $x_n > 0$ für alle $n \in \mathbb{N}$.
2. Es gilt $x_{n+1}^2 \ge a$ für alle n, denn

$$\begin{aligned} x_{n+1}^2 - a &= \frac{1}{4}\left(x_n^2 + 2a + \frac{a^2}{x_n^2}\right) - a \\ &= \frac{1}{4}\left(x_n^2 - 2a + \frac{a^2}{x_n^2}\right) \\ &= \frac{1}{4}\left(x_n - \frac{a}{x_n}\right)^2 \\ &\ge 0. \end{aligned}$$

3. Es gilt $x_{n+2} \le x_{n+1}$ für alle n, denn

$$\begin{aligned} x_{n+1} - x_{n+2} &= x_{n+1} - \frac{1}{2}\left(x_{n+1} + \frac{a}{x_{n+1}}\right) \\ &= \frac{1}{2x_{n+1}}\left(x_{n+1}^2 - a\right) \\ &\ge 0. \end{aligned}$$

4. Setze $y_n := \frac{a}{x_n}$. Dann gilt $y_{n+1}^2 \le a$ für alle n, denn nach (2) ist $\frac{1}{x_{n+1}^2} \le \frac{1}{a}$, also auch

$$y_{n+1}^2 = \frac{a^2}{x_{n+1}^2} \le \frac{a^2}{a} = a.$$

5. Aus (3) folgt $y_{n+1} \le y_{n+2}$ für alle n.

6. Es gilt $y_{n+1} \leq x_{m+1}$ für alle $n, m \in \mathbb{N}$. Denn – etwa für $n \geq m$ – hat man $y_{n+1} \leq x_{n+1}$ (dies folgt aus (2) durch Multiplikation mit $\frac{1}{x_{n+1}}$), und $x_{n+1} \leq x_{m+1}$ nach (3).

7. Es gilt

$$x_{n+1} - y_{n+1} \leq \frac{1}{2^n}(x_1 - y_1).$$

Wir zeigen dies durch Induktion über n. Induktionsanfang: Für $n = 0$ sind beide Seiten gleich. Induktionsschritt:

$$\begin{aligned} x_{n+2} - y_{n+2} &\leq x_{n+2} - y_{n+1} \\ &= \frac{1}{2}(x_{n+1} + y_{n+1}) - y_{n+1} \\ &= \frac{1}{2}(x_{n+1} - y_{n+1}) \\ &\leq \frac{1}{2^{n+1}}(x_1 - y_1) \qquad \text{nach Induktionsvoraussetzung.} \end{aligned}$$

8. $(x_n)_{n\in\mathbb{N}}$ ist CAUCHYfolge, denn für $n + 1 \leq m + 1$ gilt nach (3), (6) und (7)

$$|x_{n+1} - x_{m+1}| = x_{n+1} - x_{m+1} \leq x_{n+1} - y_{n+1} \leq \frac{1}{2^n}(x_1 - y_1).$$

9. Nehmen wir jetzt $\lim x_n = b$ an, also

$$\forall \varepsilon{>}0 \exists N{\in}\mathbb{N} \forall n{\geq}N.\, |x_n - b| \leq \varepsilon.$$

Wir zeigen zunächst $\lim y_n = b$. Sei also $\varepsilon > 0$. Dann gilt für alle $n \geq N(\frac{\varepsilon}{2})$

$$\begin{aligned} |b - y_{n+1}| &\leq |b - x_{n+1}| + |x_{n+1} - y_{n+1}| \\ &\leq \frac{\varepsilon}{2} + \frac{1}{2^n}(x_1 - y_1) \qquad \text{nach (6) und (7)} \\ &\leq \frac{\varepsilon}{2} + \frac{1}{n}(x_1 - y_1) \\ &\leq \varepsilon, \qquad \text{falls noch } n \geq \frac{2}{\varepsilon}(x_1 - y_1). \end{aligned}$$

Wegen $y_{n+1}^2 \leq a \leq x_{n+1}^2$ folgt

$$b^2 = (\lim y_n)^2 = \lim y_n^2 \leq a \leq \lim x_n^2 = (\lim x_n)^2 = b^2,$$

also $b^2 = a$. □

Um dieses NEWTON-Verfahren zu implementieren, definieren wir

```
(define (average x y)
  (/ (+ x y) 2))
```

Das im Satz formulierte Iterationsverfahren wird nun wie folgt implementiert.

```
(define (sqrt x)
  (sqrt-iter 1 x))

(define (sqrt-iter guess x)
```

```
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x)))

(define (good-enough? guess x)
  (< (abs (- (square guess) x)) .001))

(define (improve guess x)
  (average guess (/ x guess)))
```

Hier wurden alle diese Prozeduren `sqrt`, `sqrt-iter`, `good-enough` und `improve` zu der globalen Umgebung hinzugefügt. Eine Alternative besteht darin, die Hilfsprozeduren `sqrt-iter`, `good-enough` und `improve` nach außen unsichtbar zu machen und nur innerhalb der Definition von `sqrt` bereitzustellen. Das läßt sich wie folgt erreichen.

```
(define (sqrt x)
  (define (good-enough? guess x)
    (< (abs (- (square guess) x)) .001))
  (define (improve guess x)
    (average guess (/ x guess)))
  (define (sqrt-iter guess x)
    (if (good-enough? guess x)
        guess
        (sqrt-iter (improve guess x) x)))
  (sqrt-iter 1 x))
```

Hierbei läßt sich noch eine weitere Vereinfachung durchführen. Die Variable `x` wird in `sqrt` gebunden. Da die Definitionen von `good-enough`, `improve` und `sqrt-iter` sich im Bindungsbereich (scope) von `x` befinden, muß die Variable `x` nicht noch einmal explizit als Parameter übergeben werden. Man spricht hier von einem „lexikalischen Bindungsbereich“ (lexical scoping).

```
(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) .001))
  (define (improve guess)
    (average guess (/ x guess)))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve guess))))
  (sqrt-iter 1))
```

Einige Beispiele:

```
(sqrt 2) ==> 577/408
```

```
(sqrt 2.0 ) ==> 1.4142156862745097

(sqrt 9) ==> 65537/21845
(sqrt 9.0) ==> 3.00009155413138
```

Bemerkung. Man kann die internen Verwendungen von `define` – wie in der letzten Definition von `sqrt` – immer durch `letrec` ersetzen. Die allgemeine Verwendung von `let`, `let*` und `letrec` ist in der Sprachdefinition [**3**], Abschnitt 4.2.2 erklärt.

```
(define (sqrt x)
  (letrec ((good-enough?
             (lambda (guess)
                (< (abs (- (square guess) x)) .001)))
            (improve
             (lambda (guess) (average guess (/ x guess))))
            (sqrt-iter
             (lambda (guess)
               (if (good-enough? guess)
                   guess
                   (sqrt-iter (improve guess))))))
    (sqrt-iter 1)))
```

Eine Einschränkung ist bei der Verwendung von `letrec` immer zu beachten: es muß möglich sein, die rechten Seiten der Bindungspaare in `letrec` auszuwerten, ohne auf Werte der in den linken Seiten gebundenen Variablen zuzugreifen. Dies ist bei der meist vorkommenden Verwendung, in der die rechten Seiten der Bindungspaare `lambda`-Ausdrücke sind, automatisch der Fall.

2. Prozeduren und die von ihnen erzeugten Prozesse

2.1. Primitive Rekursion und Iteration. Als Beispiel für primitive Rekursion (auch lineare Rekursion genannt) betrachten wir die Definition der Fakultätsfunktion.

$$0! = 1,$$
$$(n+1)! = (n+1) \cdot n!.$$

Eine direkte Übertragung dieser Definition in die Sprache von Scheme liefert

```
(define (factorial n)
  (if (= 0 n)
      1
      (* n (factorial (- n 1)))))
```

Eine iterative Form dieser Definition ist

```
(define (factorial n)
```

```
  (fact-iter 1 1 n))

(define (fact-iter product counter max-count)
  (if (> counter max-count)
      product
      (fact-iter (* counter product)
                 (+ counter 1)
                 max-count)))
```

In `product` wird also das Ergebnis angesammelt. Man nennt deshalb ein solches zusätzliches Argument einen *Akkumulator*.

2.2. Baumrekursion. Eine weitere häufig auftretende Form der Rekursion ist die sogenannte ungeschachtelte Rekursion (auch Baumrekursion genannt). Hier dürfen mehrere ungeschachtelte Aufrufe der zu definierenden Funktion vorkommen. Ein typisches Beispiel dafür ist die Folge der Fibonacci-Zahlen:

$$\mathsf{Fib}(n) := \begin{cases} 0 & \text{falls } n = 0, \\ 1 & \text{falls } n = 1, \\ \mathsf{Fib}(n-1) + \mathsf{Fib}(n-2) & \text{sonst.} \end{cases}$$

Hieraus erhalten wir unmittelbar die folgende Definition in Scheme:

```
(define (fib n)
  (cond ((= 0 n) 0)
        ((= 1 n) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```

Man beachte jedoch, daß eine hiernach durchgeführte Berechnung sehr ineffizient ist, da viele Mehrfachberechnungen durchgeführt werden. Deshalb ist die folgende iterative Version vorzuziehen.

```
(define (fib n)
  (fib-iter 1 0 n))

(define (fib-iter res prev count)
  (if (= 0 count)
      prev
      (fib-iter (+ res prev) res (- count 1))))
```

Ein weiteres Beispiel für eine ungeschachtelte Rekursion liefern die Binomialkoeffizienten. Hier tritt als zusätzliche Besonderheit auf, daß die Parameterwerte verändert werden. Für natürliche Zahlen n und k setzen wir

$$\binom{n}{k} := \prod_{j=1}^{k} \frac{n-j+1}{j} = \frac{n(n-1) \cdot \ldots \cdot (n-k+1)}{1 \cdot 2 \cdot \ldots \cdot k}.$$

Die Zahlen $\binom{n}{k}$ heißen *Binomialkoeffizienten*.

Aus der Definition folgt unmittelbar

$$\binom{n}{k} = 0 \quad \text{für } k > n,$$

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \binom{n}{n-k} \quad \text{für } 0 \leq k \leq n.$$

LEMMA 2.1. *Für $1 \leq k \leq n$ gilt*

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

BEWEIS. Für $k = n$ ist dies offenbar richtig. Für $1 \leq k \leq n-1$ hat man

$$\begin{aligned}\binom{n-1}{k-1} + \binom{n-1}{k} &= \frac{(n-1)!}{(k-1)!(n-k)!} + \frac{(n-1)!}{k!(n-k-1)!} \\ &= \frac{k(n-1)! + (n-k)(n-1)!}{k!(n-k)!} \\ &= \frac{n!}{k!(n-k)!} \\ &= \binom{n}{k}.\end{aligned}$$

Das war zu zeigen. □

2.3. Geschachtelte Rekursion. Eine weitere häufig auftretende Form der Rekursion ist die sogenannte geschachtelte Rekursion, in der mehrere geschachtelte Aufrufe der zu definierenden Funktion vorkommen. Ein typisches Beispiel ist die ACKERMANN-Funktion, die wie folgt definiert ist.

$$\begin{aligned} f(x,0) &= 0, \\ f(0,y+1) &= 2(y+1), \\ f(x+1,1) &= 2, \\ f(x+1,y+2) &= f(x, f(x+1,y+1)). \end{aligned}$$

```
(define (ack x y)
  (cond ((= y 0) 0)
        ((= x 0) (* 2 y))
        ((= y 1) 2)
        (else (ack (- x 1) (ack x (- y 1))))))
```

ÜBUNG 2.2. Man berechne `(ack 1 10)`, `(ack 2 4)` und `(ack 3 3)`. Man verfolge den Ablauf der Rechnung mittels `(trace ack)`. Ferner gebe man übliche mathematische Definitionen für die Funktionen $f_i(y) := f(i,y)$ für $i = 0, 1, 2$.

2.4. Höherstufige Prozeduren. Betrachten wir die folgenden drei Prozeduren. Die erste berechnet die Summe aller ganzen Zahlen zwischen `a` und `b`.

```
(define (sum-integers a b)
  (if (> a b)
      0
      (+ a (sum-integers (+ a 1) b))))
```

Die zweite berechnet die Summe aller Quadrate der ganzen Zahlen zwischen `a` und `b`.

```
(define (sum-sqares a b)
  (if (> a b)
      0
      (+ (square a) (sum-squares (+ a 1) b))))
```

Die dritte berechnet gewisse Partialsummen der (sehr langsam) gegen $\frac{\pi}{8}$ konvergenten Reihe

$$\frac{1}{1\cdot 3}+\frac{1}{5\cdot 7}+\frac{1}{9\cdot 11}+\cdots,$$

und zwar bei Eingabe von $a = 4i - 3$ und $b = 4j - 3$ (mit $i \leq j$) die Partialsumme vom i-ten bis zum j-ten Glied (einschließlich).

```
(define (pi-sum a b)
  (if (> a b)
      0
      (+ (/ 1 (* a (+ a 2))) (pi-sum (+ a 4) b))))
```

Man erhält zum Beispiel

```
(* 8.0 (pi-sum 1 100)) ==> 3.1215946525910105
(* 8.0 (pi-sum 1 1500)) ==> 3.1402593208490512
```

Alle drei Definitionen folgen offenbar demselben Schema. Es liegt deshalb nahe, diese gemeinsame Form zu abstrahieren und einen allgemeinen Summationsoperator wie folgt zu definieren.

```
(define (sum summand-fct next-fct a b)
  (if (> a b)
      0
      (+ (summand-fct a)
         (sum summand-fct next-fct (next-fct a) b))))
```

Man beachte, daß hier Prozeduren als Argumente übergeben werden. In diesem Sinn ist also der definierte Summenoperator höherstufig.

Übung 2.3. Man bringe die rekursive `sum`-Definition in iterative Form. Dazu fülle man die offenen Stellen in folgendem Muster aus.

```
(define (sum summand-fct next-fct a b)
  (define (iter arg result)
    (if ??
        ??
        (iter ??
              ??)))
```

```
(iter ?? ??))
```

KAPITEL 2

Daten

1. Datenabstraktion

1.1. Arithmetische Operationen für rationale Zahlen. Nehmen wir zunächst an, wir hätten schon eine Implementierung der rationalen Zahlen durchgeführt, und zwar durch Angabe eines *Konstruktors* `make-rat`, der aus Zähler und Nenner eine rationale Zahl konstruiert, und zweier *Selektoren* `numer` und `denom`, die aus einer rationalen Zahl den Zähler bzw. den Nenner ablesen. Ohne diese Implementierung genauer zu kennen, können wir Addition, Subtraktion, Multiplikation, Division und Gleichheit rationaler Zahlen definieren:

```
(define (+rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (denom x) (numer y)))
            (* (denom x) (denom y))))

(define (-rat x y)
  (make-rat (- (* (numer x) (denom y))
               (* (denom x) (numer y)))
            (* (denom x) (denom y))))

(define (*rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))

(define (/rat x y)
  (make-rat (* (numer x) (denom y))
            (* (denom x) (numer y))))

(define (=rat x y)
  (= (* (numer x) (denom y))
     (* (numer y) (denom x))))
```

Wir wollen jetzt den Konstruktor `make-rat` und die beiden Selektoren `numer` und `denom` implementieren. Dazu benötigen wir offenbar einen Weg, aus zwei Datenobjekten – hier Zähler und Nenner – ein neues zusammenzusetzen. Eine solche Paarbildung ist die Grundform zur Bildung zusammengesetzter Daten in LISP. Zur Bildung eines Paares aus zwei Argumenten verwenden wir die primitive Prozedur `cons`. Aus einem Paar kann man die erste und die zweite Komponente ablesen mittels der primitive Prozeduren car und cdr. Die Bezeichnungen car und cdr gehen zurück auf die ursprüngliche Implementierung von LISP auf einer IBM 704. car steht für "contents of address register" und cdr steht für "contents of decrement register".

```
(define x (cons 1 2)) ==> x
(car x) ==> 1
(cdr x) ==> 2
```

Man beachte, daß ein Paar ein gewöhnliches Datenobjekt ist, das wie jedes andere mit einem Namen versehen und manipuliert werden kann.

```
(define x (cons 1 2)) ==> x
(define y (cons 3 4)) ==> y
(define z (cons x y)) ==> z
(car (car z)) ==> 1
(car (cdr z)) ==> 3
```

In LISP verwendet man die Paarbildung als universellen Baustein zur Bildung komplexer Datenobjekte.

Wir können jetzt sehr leicht rationale Zahlen implementieren.

```
(define (make-rat n d) (cons n d))
(define (numer x) (car x))
(define (denom x) (cdr x))
```

Zum Ausdruck der rationalen Zahlen verwenden wir die folgende Prozedur.

```
(define (print-rat x)
  (newline)
  (display (numer x))
  (display "/")
  (display (denom x)))
```

Dann erhält man zum Beispiel

```
(print-rat (make-rat 2 6)) ==> 2/6
```

1.2. Abstraktionsschranken. Man beachte, daß wir in unserer Implementierung der Operationen auf rationalen Zahlen vorausgesetzt haben, daß wir den Konstruktor make-rat und die Selektoren numer und denom zur Vefügung haben. Es war nicht nötig, diese Implementierung genauer zu kennen. Das einzige, das wir über den Konstruktor make-rat und die Selektoren numer und denom wissen mußten, war, daß die Anwendung eines Selektors auf ein durch den Konstruktor gebildetes Objekt die entsprechende Komponente reproduziert.

Ähnlich ist es mit `cons`, `car` und `cdr`. Eine alternative Implementierung von ihnen wäre etwa

```
(define (new-cons x y)
  (define (dispatch m)
    (cond ((= 0 m) x)
          ((= 1 m) y)
          (else (error "Argument not 0 or 1 - NEW-CONS" m))))
  dispatch)

(define (new-car z) (z 0))
(define (new-cdr z) (z 1))
```

Man beachte, daß der von `(new-cons x y)` zurückgegebene Wert eine Prozedur ist.

1.3. Konvergente Folgen rationaler Zahlen. Im folgenden benutzen wir der Einfachheit halber wieder die SCHEME-interne Darstellung der rationalen Zahlen (andernfalls müßten wir zu viele Hilfsmittel bereitstellen).

Es sei M eine beliebige Menge, etwa die Menge $\mathbb{Q}$ der rationalen Zahlen oder die Menge $\mathbb{N}$ der natürlichen Zahlen. Unter einer *Folge* von Elementen aus M versteht man eine Abbildung $\mathbb{N} \to M$. Jedem $n \in \mathbb{N}$ ist also ein $a_n \in M$ zugeordnet. Man schreibt hierfür $(a_n)_{n\in\mathbb{N}}$ oder $(a_0, a_1, a_2, \ldots)$.

Beispiele:

1. Sei $a_n = a$ für alle $n \in \mathbb{N}$. Man erhält die konstante Folge
$$(a, a, a, a, \ldots).$$
2. Sei $a_n = \frac{1}{n}$ für alle $n \geq 1$. Man erhält die Folge
$$(1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \ldots).$$
3. $a_n = \frac{n}{n+1}$:
$$(0, \frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \frac{4}{5}, \ldots).$$
4. $a_n = \frac{n}{2^n}$:
$$(0, \frac{1}{2}, \frac{2}{4}, \frac{3}{8}, \frac{4}{16}, \frac{5}{32}, \ldots).$$

Sei $(a_n)_{n\in\mathbb{N}}$ eine Folge rationaler Zahlen. Die Folge heißt *konvergent* gegen $a \in \mathbb{Q}$ (in Zeichen $\lim_{n\to\infty} a_n = a$ oder kurz $\lim a_n = a$), falls gilt:
$$\forall \varepsilon{>}0 \exists N{\in}\mathbb{N} \forall n{\geq}N.\, |a_n - a| \leq \varepsilon.$$
Eine vollständige, explizite Angabe einer gegen $a \in \mathbb{Q}$ konvergenten Folge rationaler Zahlen besteht demnach aus den folgenden Daten:

1. der Folge $(a_n)_{n\in\mathbb{N}}$,
2. der Zahl a, und
3. einer Abbildung $N\colon \mathbb{Q}^*_+ \to \mathbb{N}$, die jedem $\varepsilon > 0$ eine natürliche Zahl $N(\varepsilon)$ zuordnet, von der ab $|a_n - a|$ kleiner oder gleich ε ist; diese Abbildung nennt man *Konvergenzmodul*.

Entsprechend implementieren wir jetzt Folgen rationaler Zahlen, die gegen eine rationale Zahl konvergieren:

```
(define (make-conv s l m) (cons s (cons l m)))
(define (seq x) (car x))
(define (lim x) (car (cdr x)))
(define (mod x) (cdr (cdr x)))
```

Behandlung der Beispiele:

1. Wenn $a_n = a$ für alle $n \in \mathbb{N}$, so gilt $\lim a_n = a$. Denn sei $\varepsilon > 0$ beliebig vorgeben. Dann gilt für alle $n \in \mathbb{N}$
$$|a_n - a| = 0 \leq \varepsilon.$$
2. Es gilt $\lim_{n\to\infty} \frac{1}{n} = 0$. Denn sei $\varepsilon > 0$ beliebig vorgeben. Dann gilt
$$\left|\frac{1}{n} - 0\right| = \frac{1}{n} \leq \varepsilon \qquad \text{für alle } n \geq \tfrac{1}{\varepsilon}.$$
3. $\lim_{n\to\infty} \frac{n}{n+1} = 1$: Sei $\varepsilon > 0$ beliebig vorgeben. Dann gilt
$$\left|\frac{n}{n+1} - 1\right| = \frac{1}{n+1} \leq \varepsilon \qquad \text{für alle } n \geq \tfrac{1}{\varepsilon}.$$
4. $\lim_{n\to\infty} \frac{n}{2^n} = 0$: Es gilt $n^2 \leq 2^n$ für $n \geq 4$, wie man durch Induktion über n leicht beweist. Sei $\varepsilon > 0$ beliebig vorgeben. Dann gilt
$$\left|\frac{n}{2^n} - 0\right| = \frac{n}{2^n} \leq \frac{1}{n} \leq \varepsilon \qquad \text{für alle } n \geq \max(4, \tfrac{1}{\varepsilon}).$$

Wir wollen jetzt die Beispielsfolgen implementieren.

```
(define ex1
  (make-conv (lambda (n) 27)
             27
             (lambda (eps) 0)))
```

Für das die restlichen Beispiele ergibt sich

```
(define ex2
  (make-conv (lambda (n) (/ 1 n))
             0
             (lambda (eps) (ceiling (/ 1 eps)))))

(define ex3
  (make-conv (lambda (n) (/ n (+ n 1)))
             1
             (lambda (eps) (ceiling (/ 1 eps)))))

(define ex4
  (make-conv (lambda (n) (/ n (expt 2 n)))
             0
             (lambda (eps) (ceiling (max 4 (/ 1 eps))))))
```

Zum Beispiel hat man dann

```
((seq ex4) 7) ==> 0.0546875
((mod ex3) (/ 1 17)) ==> 17.0
```

1.4. Cauchyfolgen. Als weiteres Beispiel zur Datenabstraktion und auch zum Auftreten von Prozeduren als Werten behandeln wir die übliche Darstellung der reellen Zahlen als CAUCHYfolgen von rationalen Zahlen und einige Operationen auf solchen CAUCHYfolgen.

Bekanntlich sind die rationalen Zahlen unvollständig im folgenden Sinn.

THEOREM 1.1 (Irrationalität von $\sqrt{2}$). *Es gibt keine rationale Zahl b mit $b^2 = 2$.*

BEWEIS. Sei ein solches $b = \frac{n}{m} \in \mathbb{Q}$ gegeben, also $n^2 = 2m^2$. Die Anzahl der Primfaktoren 2 in n^2 ist gerade, dagegen in $2m^2$ ungerade. Man erhält also einen Widerspruch gegen die Eindeutigkeit der Primfaktorzerlegung natürlicher Zahlen. □

Trotzdem kann man – wie im Satz 1.5 bewiesen – nach dem NEWTON-Verfahren eine Folge rationaler Zahlen konstruieren, deren Quadrate gegen 2 konvergieren. Wir führen deshalb den Begriff der CAUCHYfolge ein. Eine Folge $(a_n)_{n\in\mathbb{N}}$ rationaler Zahlen heißt CAUCHY*folge*, wenn gilt

$$\forall\varepsilon{>}0\exists N{\in}\mathbb{N}\forall n, m{\geq}N.\, |a_n - a_m| \leq \varepsilon.$$

Eine vollständige, explizite Angabe einer CAUCHYfolge rationaler Zahlen besteht demnach aus den folgenden Daten:

1. der Folge $(a_n)_{n\in\mathbb{N}}$, und
2. einer Abbildung $N\colon \mathbb{Q}^*_+ \to \mathbb{N}$, die jedem $\varepsilon > 0$ die natürliche Zahl $N(\varepsilon)$ zuordnet, von der ab $|a_n - a_m| \leq \varepsilon$ ist; diese Abbildung nennt man CAUCHY*modul*.

Entsprechend implementieren wir jetzt CAUCHYfolgen rationaler Zahlen:

```
(define (make-cauchy s m) (cons s m))
(define (cauchy-to-seq x) (car x))
(define (cauchy-to-mod x) (cdr x))
```

THEOREM 1.2. *Jede konvergente Folge $(a_n)_{n\in\mathbb{N}}$ ist* CAUCHY*folge.*

BEWEIS. Sei $(a_n)_{n\in\mathbb{N}}$ eine gegen a konvergente Folge, also

$$\forall\varepsilon{>}0\exists N{\in}\mathbb{N}\forall n{\geq}N.\, |a_n - a| \leq \varepsilon.$$

Zu zeigen ist, daß $(a_n)_{n\in\mathbb{N}}$ CAUCHYfolge ist. Sei also $\varepsilon > 0$. Mit $M := N(\frac{\varepsilon}{2})$ gilt dann für alle $n, m \geq M$

$$\begin{aligned} |a_n - a_m| &\leq |a_n - a| + |a - a_m| \\ &\leq \frac{\varepsilon}{2} + \frac{\varepsilon}{2}. \end{aligned}$$

Also ist $(a_n)_{n\in\mathbb{N}}$ CAUCHYfolge. □

Man kann also aus den oben behandelten Beispielen konvergenter Folgen in systematischer Weise Beispiele für CAUCHYfolgen erhalten.

ÜBUNG 1.3. Man führe dies aus. Außerdem gebe man direkt CAUCHY-moduln für die Beispielsfolgen an.

Für CAUCHYfolgen rationaler Zahlen, die nicht gegen eine rationale Zahl konvergieren, kann man einen CAUCHYmodul nicht so einfach gewinnen. Hier muß man den Beweis, daß es sich um eine CAUCHYfolge handelt, genauer analysieren. Betrachten wir noch einmal die nach dem NEWTON-Verfahren konstruierte Folge rationaler Zahlen, deren Quadrate gegen 2 konvergieren. Unter Verwendung der oben (in Theorem 1.5) definierten Funktion `improve` können wir diese Folge implementieren durch

```
(define (sqrt-2-seq n)
  (if (= 0 n)
      1
      (improve (sqrt-2-seq (- n 1)) 2)))
```

Man erhält

```
(sqrt-2-seq 1)  ==> 1.5
(sqrt-2-seq 2)  ==> 1.41666666666667
(sqrt-2-seq 3)  ==> 1.41421568627451
(sqrt-2-seq 4)  ==> 1.41421356237469
(sqrt-2-seq 5)  ==> 1.41421356237309
(sqrt-2-seq 6)  ==> 1.41421356237309
```

Wenn man jedoch die reelle Zahl $\sqrt{2}$ in unserem Sinn vollständig angeben will, braucht man noch einen Cauchymodul.

ÜBUNG 1.4. Man lese aus dem Beweis des zum NEWTON-Verfahren gehörigen Satzes 1.5 einen CAUCHYmodul für die Folge der Approximationen von $\sqrt{2}$ mit Startwert 1 ab und implementiere ihn.

Ferner betrachten wir den Begriff der Beschränktheit von Folgen. Eine Folge rationaler Zahlen heißt *nach oben* (bzw. *nach unten*) *beschränkt*, wenn es ein $K \in \mathbb{Q}$ gibt, so daß $a_n \leq K$ (bzw. $a_n \geq K$) für alle $n \in \mathbb{N}$. Die Folge heißt *beschränkt*, wenn sie nach oben und nach unten beschränkt ist.

THEOREM 1.5. *Jede CAUCHYfolge $(a_n)_{n\in\mathbb{N}}$ ist beschränkt.*

BEWEIS. Sei $(a_n)_{n\in\mathbb{N}}$ CAUCHYfolge, also

$$\forall\varepsilon{>}0\exists N{\in}\mathbb{N}\forall n, m{\geq}N.\, |a_n - a_m| \leq \varepsilon.$$

Wählt man 1 für ε, so gilt für alle $n, m \geq N$

$$|a_n - a_m| \leq 1.$$

Mit

$$K := (\max_{0\leq k\leq N} a_n) + 1$$

haben wir deshalb $a_n \leq K$ für alle n. Denn für $n \leq N$ gilt dies nach Definition von K, und für $n \geq N$ hat man

$$a_n \leq a_N + |a_n - a_N| \leq K.$$

Damit ist gezeigt, daß $(a_n)_{n\in\mathbb{N}}$ nach oben beschränkt ist. Da mit $(a_n)_{n\in\mathbb{N}}$ auch $(-a_n)_{n\in\mathbb{N}}$ CAUCHYfolge ist, kann man wie eben auch eine obere Schranke L für $(-a_n)_{n\in\mathbb{N}}$ finden. Dann ist $-L$ eine untere Schranke für $(a_n)_{n\in\mathbb{N}}$. □

Wir können den Beweis dieses Satzes benutzen, um die Berechnung einer oberen Schranke für eine (durch Folge und Modul gegebene) CAUCHYfolge zu implementieren.

```
(define (cauchy-to-bound x)
  (letrec ((seq (cauchy-to-seq x))
           (mod (cauchy-to-mod x))
           (seq-to-max
             (lambda (s a b)
               (if (> a b)
                   0
                   (max (s a) (seq-to-max s (+ a 1) b))))))
    (+ (seq-to-max seq 0 (mod 1)) 1)))
```

THEOREM 1.6 (Summe von CAUCHYfolgen). *Seien $(a_n)_{n\in\mathbb{N}}$ und $(b_n)_{n\in\mathbb{N}}$ CAUCHYfolgen rationaler Zahlen. Dann ist auch die Folge $(c_n)_{n\in\mathbb{N}}$ mit $c_n := a_n + b_n$ eine CAUCHYfolge.*

BEWEIS. Sei $\varepsilon > 0$. Dann ist auch $\frac{\varepsilon}{2} > 0$. Da $(a_n)_{n\in\mathbb{N}}$ CAUCHYfolge ist, gibt es ein $N \in \mathbb{N}$ mit

$$|a_n - a_m| \leq \frac{\varepsilon}{2} \qquad \text{für alle } n, m \geq N.$$

Da auch $(b_n)_{n\in\mathbb{N}}$ CAUCHYfolge ist, gibt es ein $M \in \mathbb{N}$ mit

$$|b_n - b_m| \leq \frac{\varepsilon}{2} \qquad \text{für alle } n, m \geq M.$$

Für alle $n, m \geq \max(N, M)$ gilt also

$$|(a_n + b_n) - (a_m + b_m)| \leq |a_n - a_m| + |b_n - b_m| \leq \frac{\varepsilon}{2} + \frac{\varepsilon}{2} = \varepsilon.$$

Daher ist die Folge $(a_n + b_n)$ eine CAUCHYfolge. □

ÜBUNG 1.7. Man implementiere die Konstruktion der Summe zweier CAUCHYfolgen.

THEOREM 1.8 (Produkt von CAUCHYfolgen). *Seien $(a_n)_{n\in\mathbb{N}}$ und $(b_n)_{n\in\mathbb{N}}$ CAUCHYfolgen rationaler Zahlen. Dann ist auch die Folge $(c_n)_{n\in\mathbb{N}}$ mit $c_n := a_n b_n$ eine CAUCHYfolge.*

BEWEIS. Nach einem früheren Satz sind die CAUCHYfolgen $(a_n)_{n\in\mathbb{N}}$ und $(b_n)_{n\in\mathbb{N}}$ beschränkt. Es gibt also $K > 0$ und $L > 0$ mit $|a_n| \leq K$ und $|b_n| \leq L$ für alle $n \in \mathbb{N}$.

Sei $\varepsilon > 0$. Da $(a_n)_{n\in\mathbb{N}}$ CAUCHYfolge ist, gibt es ein $N(\frac{\varepsilon}{2L}) \in \mathbb{N}$ mit

$$|a_n - a_m| \leq \frac{\varepsilon}{2L} \qquad \text{für alle } n, m \geq N(\frac{\varepsilon}{2L}).$$

Da auch $(b_n)_{n\in\mathbb{N}}$ CAUCHYfolge ist, gibt es ein $M(\frac{\varepsilon}{2K}) \in \mathbb{N}$ mit

$$|b_n - b_m| \leq \frac{\varepsilon}{2K} \qquad \text{für alle } n, m \geq M(\frac{\varepsilon}{2K}).$$

Für alle $n, m \geq \max(N(\frac{\varepsilon}{2L}), M(\frac{\varepsilon}{2K}))$ gilt also

$$\begin{aligned}
|a_n b_n - a_m b_m| &= |a_n(b_n - b_m) + (a_n - a_m)b_m| \\
&\leq |b_n - b_m||a_n| + |a_n - a_m||b_m| \\
&\leq |b_n - b_m|K + |a_n - a_m|L \\
&\leq \frac{\varepsilon}{2K}K + \frac{\varepsilon}{2L}L \\
&= \varepsilon.
\end{aligned}$$

Daher ist die Folge $(a_n b_n)$ eine CAUCHYfolge. ∎

ÜBUNG 1.9. Man implementiere die Konstruktion des Produktes zweier CAUCHYfolgen.

THEOREM 1.10 (Quotienten von CAUCHYfolgen). *Seien $(a_n)_{n\in\mathbb{N}}$ und $(b_n)_{n\in\mathbb{N}}$ CAUCHYfolgen rationaler Zahlen. Ferner gebe es ein $\delta > 0$ und ein $n_0 \in \mathbb{N}$ mit $\delta \leq |b_n|$ für alle $n \geq n_0$. Dann ist auch die Folge $(\frac{a_n}{b_n})_{n\in\mathbb{N}}$ eine CAUCHYfolge.*

BEMERKUNG. $\frac{a_n}{b_n}$ kann hier im Fall $b_n = 0$ beliebig festgesetzt sein, etwa als 0.

BEWEIS. Wir behandeln zunächst den Spezialfall, daß (a_n) die konstante Folge $a_n = 1$ ist.

$(b_n)_{n\in\mathbb{N}}$ sei CAUCHYfolge, also

$$\forall\varepsilon{>}0\exists N{\in}\mathbb{N}\forall n, m{\geq}N. |b_n - b_m| \leq \varepsilon.$$

Ferner seien $\delta > 0$ und $n_0 \in \mathbb{N}$ gegeben mit $\delta \leq |b_n|$ für alle $n \geq n_0$. Dann gilt für alle $\varepsilon > 0$

$$\begin{aligned}
\left|\frac{1}{b_n} - \frac{1}{b_m}\right| &= \frac{|b_m - b_n|}{|b_n b_m|} \\
&\leq \frac{|b_n - b_m|}{\delta^2} \\
&\leq \varepsilon \qquad \text{falls } n, m \geq \max(n_0, N(\varepsilon\delta^2)).
\end{aligned}$$

Also ist auch $(\frac{1}{b_n})_{n\in\mathbb{N}}$ CAUCHYfolge.

Der allgemeine Fall läßt sich aufgrund einer früheren Satzes auf den Spezialfall zurückführen, denn es ist $\frac{a_n}{b_n} = a_n \frac{1}{b_n}$. ∎

ÜBUNG 1.11. Man implementiere die Konstruktion des Quotienten zweier CAUCHYfolgen. Dabei beachte man, daß eine untere Schranke `delta` für die Beträge der Glieder der zweiten Folge als zusätzliches Argument benötigt wird.

2. Hierarchische Daten

2.1. Darstellung von Listen. Paare werden in SCHEME hauptsächlich zur Bildung von *Listen* verwendet. Listen werden dargestellt mittels iterierter Paarbildung, wobei das `cdr`-Feld des letzten Paares leer bleibt. Genauer definiert man Listen rekursiv durch die folgenden Klauseln.

1. Die leere Liste ist eine Liste.

2. Ist a ein Datenobjekt und ℓ eine Liste, so ist das Paar, dessen `car`-Feld das Datenobjekt a und dessen `cdr`-Feld die Liste ℓ enthält, eine Liste.

Die Datenobjekte in den `car`-Feldern der zur Listenbildung verwendeten Paare sind die *Elemente* der Liste. Man beachte, daß als neues Datenobjekt die *leere Liste* () benötigt wird. Die primitive Prozedur `null?` fragt ab, ob ein Datenobjekt die leere Liste ist. Für eine genauere Diskussion und eine Beschreibung der zur Listenbearbeitung zur Verfügung stehenden Prozeduren sei auf Abschnitt 6.3 in der Sprachdefinition [**3**] verwiesen. Insbesondere sind die folgenden Prozeduren wichtig:

```
list?
list
length
append
reverse
list-ref
member
```

Ferner wird oft die (höherstufige) Prozeduren `map` verwendet. `map` nimmt eine Prozedur und eine Liste und wendet die Prozedur der Reihe nach auf alle Elemente der Liste an. Zum Beispiel ist

```
(define a (list 1 2 3 4))  ==>  a
a                          ==>  (1 2 3 4)
(map square a)             ==>  (1 4 9 16)
```

ÜBUNG 2.1. Aufgrund von Lemma 2.1 kann man die Binomialkoeffizienten mittels des PASCAL*schen Dreiecks* berechnen:

$$\begin{array}{ccccccccc} & & & & 1 & & & & \\ & & & 1 & & 1 & & & \\ & & 1 & & 2 & & 1 & & \\ & 1 & & 3 & & 3 & & 1 & \\ 1 & & 4 & & 6 & & 4 & & 1 \end{array}$$

Man schreibe unter Verwendung dieser Darstellung ein Programm zur Berechnung der Binomialkoeffizienten und diskutiere es unter Effizienzgesichtspunkten.

ÜBUNG 2.2. Man schreibe ein Programm, das berechnet, auf wie viele Arten man 1 DM in Münzen wechseln kann.

Als ein weiteres Beispiel für die Verwendung von Listen betrachten wir *Polynome*. Ein Polynom $a_n X^n + \cdots + a_1 X + a_0$ mit $a_n \neq 0$ sei als Liste $(a_0 \ \ldots \ a_n)$ dargestellt (man beachte die Reihenfolge!). Das Nullpolynom wird durch die leere Liste () repräsentiert.

ÜBUNG 2.3. Zu schreiben ist eine Funktion `poly+` von zwei Argumenten, die die Summe zweier Polynome berechnen soll.

BEMERKUNG. Man kann `poly-` analog definieren (Subtraktion von Polynomen), es ist nur das Zeichen $+$ in der Zeile `(else ... )` durch $-$ zu ersetzen.

ÜBUNG 2.4. Zu schreiben ist eine Funktion `poly*skal` von zwei Argumenten, einem Polynom p und einer Zahl a, die das Ergebnis der Multiplikation von p mit a liefern soll.

ÜBUNG 2.5. Zu schreiben ist eine Funktion `poly*` von zwei Polynom-Argumenten p und q, die das Ergebnis der Multiplikation von p und q liefert.

ÜBUNG 2.6. Zu schreiben ist eine Funktion `polydivide`, die zu ihren Argumenten p und q (Polynome) Quotient a und Rest r berechnet (d.h. $p = a \cdot q + r$ und $r = 0$ oder $\deg(r) < \deg(q)$). (Diese Aufgabe ist etwas schwieriger. Es empfiehlt sich, hier die Polynome "von hinten", d.h. beginnend mit dem Leitkoeffizienten, zu bearbeiten. Deswegen werden die Polynome erst einmal umgedreht. Außerdem ist es für die Rechnung praktischer, wenn man die Polynome so normalisiert, daß q normiert ist, d.h. Leitkoeffizient 1 hat. Der Quotient ändert sich dadurch nicht; der Rest muß am Ende wieder mit dem Leitkoeffizienten von q multipliziert werden.)

ÜBUNG 2.7. Zu schreiben ist eine Funktion `polygcd`, die den größten gemeinsamen Teiler ihrer zwei Argumente p und q (Polynome) berechnet.

Als Ergänzung noch eine Funktion zur Ausgabe von Polynomen auf den Bildschirm:

```
(define (polywrite p) ;p Polynom
  (if (null? p)
      (display 0)  ;Nullpolynom gibt 0
      (do ((n (- (length p) 1) (- n 1)) ;n aktueller Exponent
           (l (reverse p) (cdr l)) ;l Koeffizienten-Liste
                                   ;(absteigend)
           (flag #t #f)) ;Flag fuer Anfang (wegen Vorzeichen)
          ((null? l))     ;Koeffizienten abgearbeitet
          (if (not (zero? (car l))) ;Koeff.  muss /= 0 sein
              (begin (polywrite1 (car l) flag n)
                       ;gibt ein Glied aus
                     (if (not (zero? n)) ;X^0 wird weggelassen
                         (begin (display "X")
                                (if (not (= n 1)) ;X^1 wird X
                                    (begin (display "^")
                                           (display n))))))))))
  (newline))   ;Zeile abschliessen

(define (polywrite1 k flag n) ;k Zahl, n natuerliche Zahl
  (if (not flag)
      (display " ")) ;wenn nicht am Anfang, ein Leerzeichen
  (if (or (not flag) (negative? k)) ;evtl. Vorzeichen ausgeben
      (display (if (negative? k) "- " "+ ")))
  (if (or (not (= 1 (abs k))) (zero? n))
      (display (abs k)))) ;Betrag des Koeffizienten ausgeben
```

Durch Schachtelung der Listenbildung kann man auch endlich verzweigte *Bäume* darstellen.

2.2. Symbole und die Notwendigkeit der Quotierung. Bisher haben wir alle unsere Datenobjekt letzten Endes aus Zahlen konstruiert. Wir wollen uns jetzt die Möglichkeit verschaffen, auch Symbole als Datenobjekte zu verwenden. Beispiele:

```
(a b c d)
((a 1) (b 7) (c 3))
```

Listen, die auch Symbole enthalten, haben eine Form, wie sie auch bisher schon häufig vorgekommen ist.

```
(* (+ 1 2) (+ x 7))

(define (factorial n)
  (if (= 0 n) 1 (* n (factorial (- n 1)))))
```

Um mit Symbolen umgehen zu können, brauchen wir die Möglichkeit der *Quotierung*. Wenn wir zum Beispiel die Liste `(a b)` bilden wollen, können wir nicht einfach `(list a b)` auswerten, da dann der Interpreter nach Werten von `a` und `b` suchen und nicht die Symbole selbst nehmen würde. Dieses Phänomen ist aus natürlichen Sprachen gut bekannt. Wir verwenden deshalb den Quotierungoperator `quote` und schreiben `(quote a)`, wenn wir `a` quotieren wollen. Für `(quote a)` kann man kürzer auch `'a` schreiben.

```
(define a 1) ==> a
(define b 2) ==> b
(list a b)   ==> (1 2)
(list 'a 'b) ==> (a b)
(list 'a b)  ==> (a 2)
```

Auch zusammengesetzte Objekte kann man quotieren.

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

2.3. Darstellung von endlichen Mengen. Mengen kann man auf viele verschiedene Arten repräsentieren. Hierbei sind in erster Linie Effizienzgesichtspunkte zu beachten.

Wir verwenden die Methode der Datenabstraktion. Das heißt, daß wir Mengen ausschließlich mit den Operationen

`adjoin-set`, `empty-set`, `element-of-set?`, `empty-set?`,

`union-set` und `intersection-set`

bearbeiten.

Als erstes implementieren wir Mengen als *ungeordnete Listen.*

```
(define empty-set '())

(define (empty-set? set) (null? set))

(define (element-of-set? x set)
  (cond ((empty-set? set) #f)
        ((equal? x (car set)) #t)
        (else (element-of-set? x (cdr set)))))

(define (adjoin-set x set)
  (if (element-of-set? x set)
      set
      (cons x set)))

(define (intersection-set set1 set2)
  (cond ((or (empty-set? set1) (empty-set? set2)) empty-set)
        ((element-of-set? (car set1) set2)
         (cons (car set1)
               (intersection-set (cdr set1) set2)))
        (else (intersection-set (cdr set1) set2))))
```

ÜBUNG 2.8. Man implementiere entsprechend `union-set` für die Darstellung von Mengen als ungeordnete Listen.

Unter Effizienzgesichtspunkten ist diese Implementierung nicht sehr befriedigend. Die Grundoperation `element-of-set?` benötigt n Schritte, um eine Menge bestehend aus n Elementen daraufhin zu prüfen, ob das gegebene Objekt Element der Menge ist. Die benötigte Zeit wächst also wie $O(n)$, wenn n die Größe der Menge ist. Die Operation `adjoin-set`, die `element-of-set?` verwendet, braucht also auch $O(n)$ viele Schritte. Die Operation `intersection-set` verwendet für jedes Element von `set1` einen Test `element-of-set?`, braucht also insgesamt $O(n^2)$ viele Schritte. Dasselbe gilt für `union-set`.

Als nächstes implementieren wir Mengen als *geordnete Listen.* Dafür ist es notwendig, daß wir eine lineare Ordnung der Elemente gegeben haben. Man könnte etwa die lexikographische Ordnung für Symbole verwenden. Zur Vereinfachung nehmen wir hier an, daß nur ganze Zahlen als Elemente in Frage kommen.

```
(define (element-of set? x set)
  (cond ((empty-set? set) #f)
        ((= x (car set)) #t)
        ((< x (car set)) #f)
        (else (element-of set? x (cdr set)))))
```

Im Mittel benötigt man nur halb soviele Schritte wie bei der Implementierung von Mengen durch ungeordnete Listen. Dies ist aber immer noch von der Größenordnung $O(n)$.

Eine wesenlich größere Beschleunigung erhält man bei der Durchschnittsbildung:

```
(define (intersection-set set1 set2)
  (if (or (empty-set? set1) (empty-set? set2))
      empty-set
      (let ((x1 (car set1))
            (x2 (car set2)))
        (cond
          ((= x1 x2)
           (cons x1 (intersection-set (cdr set1) (cdr set2))))
          ((< x1 x2)
           (intersection-set (cdr set1) set2))
          ((> x1 x2)
           (intersection-set set1 (cdr set2)))))))
```

Diese Implementierung braucht nur noch $O(n)$ viele Schritte.

Übung 2.9. Man gebe eine entsprechende $O(n)$-Implementierung von `union-set` für die Darstellung von Mengen als geordnete Listen.

Schließlich implementieren wir Mengen als *binäre Bäume*. Hierbei wird an jeden Knoten ein Element geschrieben, und es wird verlangt, daß der linke Teilbaum nur kleinere, der rechte Teilbaum nur größere Elemente enthält. Wenn dann der Baum "ausgeglichen" (oder balanziert) ist, benötigt die Abfrage `element-of set?` nur noch $O(\log n)$ viele Schritte.

```
(define (entry tree) (car tree))
(define (left-branch tree) (cadr tree))
(define (right-branch tree) (caddr tree))
(define (make-tree entry left right) (list entry left right))
(define empty-set '())
(define (empty-set? set) (null? set))

(define (element-of set? x set)
  (cond ((empty-set? set) #f)
        ((= x (entry set)) #t)
        ((< x (entry set))
         (element-of-set? x (left-branch set)))
        ((> x (entry set))
         (element-of-set? x (right-branch set)))))
```

Das Hinzufügen eines Elements zu einer Menge wird ähnlich vorgenommen und erfordert ebenfalls $O(\log(n))$ viele Schritte.

```
(define (adjoin-set x set)
```

```
(cond ((empty-set? set) (make-tree x '() '()))
      ((= x (entry set)) set)
      ((< x (entry set))
       (make-tree (entry set)
                  (adjoin-set x (left-branch set))
                  (right-branch set)))
      ((> x (entry set))
       (make-tree (entry set)
                  (left-branch set)
                  (adjoin-set x (right-branch set))))))
```

Die Durchschnittsbildung benötigt allerdings $O(n\log(n))$ viele Schritte.

Ein Problem der Darstellung von Mengen als binäre Bäume besteht darin, daß die angegebenen Schranken für die Schrittzahlen nur dann zutreffen, wenn die Bäume tatsächlich ausgeglichen sind. Dies können wir zwar erwarten, wenn man Elemente "zufällig" hinzufügt, aber man kann natürlich nicht sicher sein. Ein Ausweg besteht darin, daß man nach jeweils einigen Hinzufügungsschritten eine Operation einschaltet, die eventuell unausgeglichene Bäume in ausgeglichene umformt.

KAPITEL 3

Zuweisungen und Umgebungen

1. Zuweisungen

Zuweisungen werden mittels der speziellen Form `set!` vorgenommen. Beispiel:

```
(define x 2)
(+ x 1)        ==> 3
(set! x 4)     ==> unspecified
(+ x 1)        ==> 5
```

2. Das Umgebungsmodell der Auswertung

Wir beschreiben das Umgebungsmodell der Auswertung anhand des folgenden Beispiels.

```
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds")))

(define f1 (make-withdraw 100))
(define f2 (make-withdraw 100))

(f1 50) ==> 50
(f2 70) ==> 30
(f2 40) ==> "Insufficient funds"
(f1 40) ==> 10
```

Hierbei haben wir die spezielle Form `begin` (auch `sequence` genannt) verwendet.

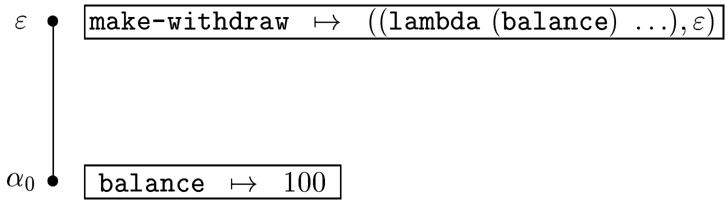
Das Resultat der Auswertung von `make-withdraw` in der globalen Umgebung ist folgendes. Die globale Umgebung wird erweitert um ein Paar

$$\texttt{make-withdraw} \quad \mapsto \quad (\texttt{(lambda (balance) ...)}, \varepsilon).$$

Das Resultat der anschließenden Auswertung von

```
(define f1 (make-withdraw 100))
```

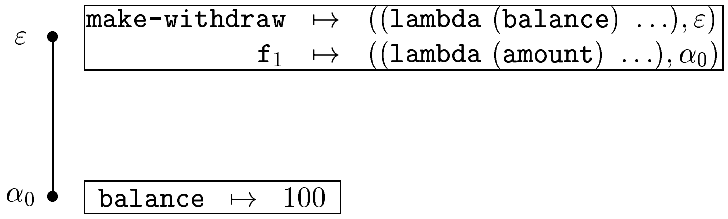
in der (erweiterten) globalen Umgebung ist folgendes. Die Auswertung von `make-withdraw` (das durch einen `lambda`-Ausdruck definiert ist) liefert einen neuen Rahmen, der der gegenwärtigen (also der globalen) Umgebung untergeordnet wird; er wird also an einem neuen Knoten $\alpha_0 := \langle 0 \rangle$ eingehängt. In diesem Rahmen ist dem Parameter `balance` der Wert 100 zugeordnet.



Mit Bezug auf den Knoten α_0 wird dann der Kern der Definition von `(make-withdraw balance)` ausgewertet, also der `lambda`-Ausdruck

```
(lambda (amount)
  (if (>= balance amount)
      (begin (set! balance (- balance amount))
             balance)
      "Insufficient funds"))
```

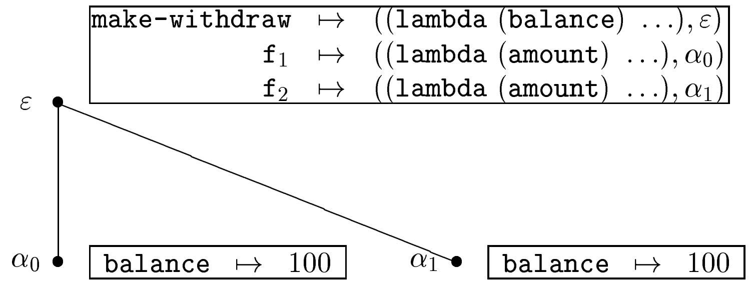
Jetzt wird ein neues Prozedurobjekt erzeugt, dessen Code der `lambda`-Ausdruck ist und dessen Zeiger auf α_0 zeigt, also auf den Knoten, an dem der `lambda`-Ausdruck ausgewertet wurde. Das entstehende Prozedurobjekt ist der Wert, der beim Aufruf von `(make-withdraw 100)` zurückgegeben wird. Er wird also an `f1` in der globalen Umgebung gebunden (denn `define` selbst wurde in der globalen Umgebung aufgerufen).



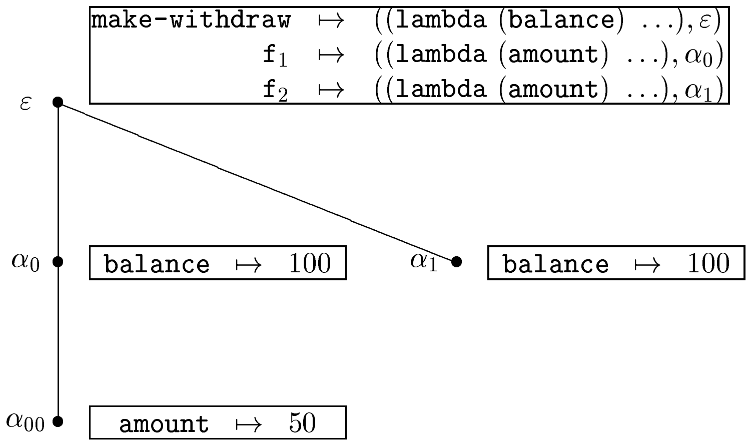
Wir wollen uns jetzt überlegen, was geschieht, wenn ein zweites Prozedurobjekt durch einen weiteren Aufruf von `make-withdraw` erzeugt wird:

```
(define f2 (make-withdraw 100))
```

Wieder wird ein neuer Rahmen der globalen Umgebung untergeordnet, also an einem neuen Knoten $\alpha_1 := \langle 1 \rangle$ eingehängt. In diesem Rahmen ist dem Parameter `balance` der Wert 100 zugeordnet.



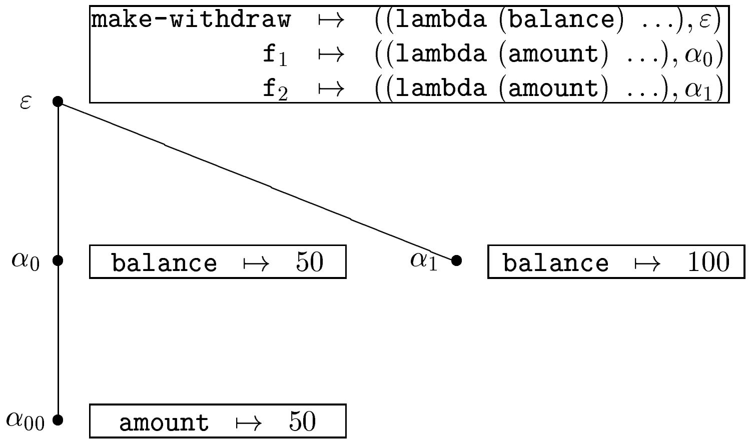
Jetzt können wir analysieren, was passiert, wenn wir anschließend `(f1 50)` aufrufen. Wieder wird ein neuer Rahmen konstruiert, in dem dem Parameter `amount` des zuerst erzeugten Prozedurobjekts `f1` der Wert 50 zugeordnet ist, und dieser Rahmen wird an einem neuen Knoten $\alpha_{00} := \langle 0, 0\rangle$ eingehängt.



Der wesentliche Punkt ist, daß dieser Rahmen also *nicht* der globalen Umgebung untergeordnet ist, sondern dem Knoten α_0, auf den der Zeiger des `f1`-Prozedurobjekts zeigt. In dieser neuen Umgebung werten wir nun den Kern des Prozedurobjekts aus, also

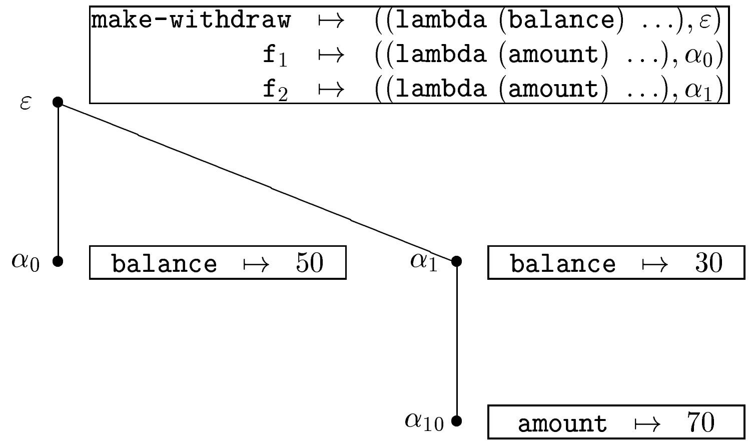
```
(if (>= balance amount)
    (begin (set! balance (- balance amount))
           balance)
    "Insufficient funds")
```

Dieser Ausdruck enthält die Symbole `balance` und `amount`. `amount` ist belegt im gerade konstruierten Rahmen am Knoten α_{00}, und `balance` wird im darüberliegenden Rahmen am Knoten α_0 gefunden. Wenn nun der `set!`-Ausdruck ausgewertet wird, verändert sich der Wert von `balance` am Knoten α_0.



Nach Abschluß des Aufrufs von `f1` ist also `balance` am Knoten α_0 an 50 gebunden, und der Zeiger des Prozedurobjekts `f1` zeigt immer noch auf α_0. Der Rahmen im Knoten α_{00}, in dem wir den Wert des obigen Ausdrucks ausgerechnet haben, wird jetzt nicht mehr benötigt. Der Aufruf, der diesen Rahmen erzeugt hat, ist nämlich abgeschlossen, und es gibt nirgendwo einen Zeiger, der auf den Knoten α_{00} dieses Rahmens zeigt. Wir werden diesen Rahmen deshalb nicht mehr aufschreiben.

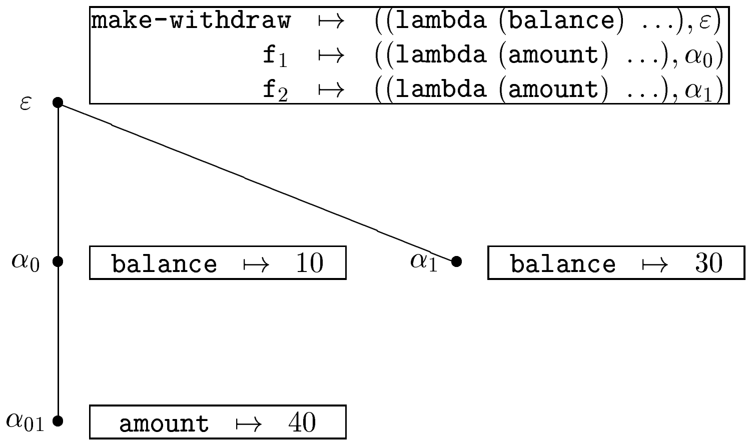
Beim jetzt folgenden Aufruf von (`f2` 70) wird dann ein neuer `amount` an 70 bindender Rahmen an einem neuen Knoten α_{10} erzeugt; er ist also dem Rahmen am Knoten α_1 untergeordnet, der die zu `f2` gehörige Bindung von `balance` enthält. Diesen Rahmen am Knoten α_1 kann man sich also vorstellen als den Ort, an dem der Wert der lokalen Zustandsvariablen `balance` des Prozedurobjekts `f2` abgelegt ist. Die Ausführung des `set!`-Ausdrucks reduziert den Wert von `balance` am Knoten α_1 auf 30.



Wieder wird der Rahmen im Knoten α_{10} nicht mehr benötigt, und wir werden ihn deshalb nicht mehr aufschreiben.

Beim anschließenden Aufruf von (`f2` 40) wird dann ein neuer `amount` an 40 bindender Rahmen an einem neuen Knoten α_{11} erzeugt, der α_1 untergeordnet ist. Jetzt liefert also der Test `(>= balance amount)` den Wert `#f` und wir erhalten die Fehlermeldung `Insufficient funds`.

Beim nächsten Aufruf von (`f1` 40) wird dann ein neuer `amount` an 40 bindender Rahmen an einem neuen Knoten α_{01} erzeugt; er ist also wieder dem Rahmen am Knoten α_0 untergeordnet, der die zu `f1` gehörige Bindung von `balance` enthält. Die Ausführung des `set!`-Ausdrucks reduziert den Wert von `balance` am Knoten α_0 auf 10.



Man beachte, daß `f1` und `f2` denselben Code besitzen, nämlich den `lambda`-Ausdruck im Kern von `make-withdraw`. Es sollte aber jetzt klar sein, warum sich `f1` und `f2` wie voneinander unabhängige Objekte verhalten. Ein Aufruf von `f1` bezieht sich auf die Belegung der Variablen `balance` am Knoten α_0, während ein Aufruf von `f2` sich auf die Belegung der Variablen `balance` am Knoten α_1 bezieht.

3. Modellierung mit veränderbaren Daten

Wir behandeln jetzt noch Modellierung mit veränderbaren Daten. Neben der Umgebung für die Variablen können auch Daten (d.h. Paare) verändert werden.

```
(define x '(a))         ==> x
(define y (cons 'b x))  ==> y
x                       ==> (a)
y                       ==> (b a)
(set-cdr! x y)          ==> unspecified


(define (switch)
  (set! x (cdr x))
  (car x))              ==> switch

(switch)                ==> b
(switch)                ==> a
(switch)                ==> b
```

```
(switch)                    ==> a
```

Man sieht, daß `x` mit einer *zyklischen Liste* belegt ist. Die Auswertung von `x` (genauer: der Versuch, `x` auszudrucken) führt zu einer Endlosschleife.

KAPITEL 4

Interpretation von Scheme in Scheme

1. Das Umgebungsmodell

Wir wollen jetzt versuchen, das im vorigen Kapitel in Abschnitt 2 angedeutete Umgebungsmodell zu präzisieren. Um die Definitionen nicht zu kompliziert werden zu lassen, beschränken wir uns auf einen repräsentativen Teil von SCHEME. Die auf diesem Umgebungsmodell basierende Semantik läßt sich leicht mit rein funktionalen Mitteln (d.h. ohne Seiteneffekte) in SCHEME programmieren.

BEMERKUNG. Die Baumstruktur der Umgebung wird in [**1**] nicht explizit gemacht. Verschiedene Rahmen können demselben darüberliegenden Rahmen untergeordnet sein. Wenn man jetzt eine Bindung in dem darüberliegenden Rahmen verändert, so betrifft dies auch alle ihm untergeordneten Rahmen. Das wird in der Darstellung von [**1**] nicht deutlich, da Umgebungen nur als Listen dargestellt sind, die dann aber intern auf dieselben Rahmen zugreifen. Eine solcher Rahmen wird dann mittels

```
(define (set-binding-value! binding value)
  (set-cdr! binding value))
```

für alle auf sie Bezug nehmenden Umgebungen geändert.

1.1. Bezeichnungen. Zunächst werden einige Notationen eingeführt.

1. $\mathbb{N} := \{0, 1, 2, \ldots\}$. Natürliche Zahlen werden durch i, j, k, l, n, m (evt. mit Indizes) mitgeteilt. $\mathbb{N}^*$ ist die Menge der endlichen Zahlenfolgen, die mit $\alpha, \beta, \ldots$ bezeichnet werden. ε sei die leere Folge. Ist $\alpha = \langle n_1, \ldots, n_k \rangle \in \mathbb{N}^*$, so sei $\alpha * \langle n_{k+1} \rangle = \langle n_1, \ldots, n_k, n_{k+1} \rangle$. $\alpha \leq \beta$ bedeute, daß α ein Anfangsstück von β ist. Ein *Baum* ist eine Teilmenge von $\mathbb{N}^*$, die abgeschlossen unter Anfangsstücken ist, und die mit $\langle n_1, \ldots, n_k, n \rangle$ stets auch alle $\langle n_1, \ldots, n_k, m \rangle$ für $m < n$ enthält.

2. Symb sei die Menge der *Symbole*.

3. Sei

$$\mathsf{atom} := \{\#t, \#f, ()\} \cup \mathbb{N} \cup \mathsf{Symb}$$

die Menge der *atomaren Datenobjekte*. Die Menge Data der *endlichen Datenobjekte* ist induktiv definiert als die Menge aller binären Bäume mit Atomen an den Blättern, also durch

1. $\mathsf{atom} \subseteq \mathsf{Data}$,
2. Sind $d_1, d_2 \in \mathsf{Data}$, so ist auch $(d_1 \ . \ d_2) \in \mathsf{Data}$.

Wir schreiben $(d_1 \ d_2 \ \ldots \ d_n)$ für $(d_1 \ . \ (d_2 \ . \ \ldots \ (d_n \ . \ ()) \ldots))$.

4. Ein "mathematisches" Paar von Objekten a und b wird wie üblich durch (a, b) mitgeteilt. Dies darf nicht mit den oben definierten Paaren

$(d_1 \;.\; d_2)$ verwechselt werden. $A \times B := \{\, (a,b) \mid a \in A, b \in B \,\}$, $\pi_i(a_1,a_2) := a_i$.

5. Eine *endliche Abbildung* ist eine Abbildung $F\colon A \to B$ mit endlichem Definitionsbereich $\mathsf{dom}(F) := A$. Sind F und G endliche Abbildungen, so sei $F[G]$ die endliche Abbildung mit $\mathsf{dom}(F[G]) = \mathsf{dom}(F) \cup \mathsf{dom}(G)$ und

$$F[G](a) := \begin{cases} G(a) & \text{falls } a \in \mathsf{dom}(G) \\ F(a) & \text{sonst,} \end{cases}$$

d.h. G überschreibt F. Offensichtlich ist diese Operation assoziativ, d.h. $(F[G])[H] = F[G[H]]$. Ist $\mathsf{dom}(F) = \{a_1, \ldots, a_n\}$, so schreiben wir statt F auch $(a_1, \ldots, a_n) \mapsto (F(a_1), \ldots, F(a_n))$.

1.2. Scheme-Ausdrücke. Sei $\mathsf{Konst} := \{\#t, \#f\} \cup \mathbb{N} \cup \{\, (\texttt{quote}\ d) \mid d \in \mathsf{Data} \,\}$ die Menge der SCHEME-*Konstanten*. Die Menge Var der SCHEME-*Variablen* sei die Menge der Symbole, welche nicht mit einer Ziffer beginnen und nicht in der Menge der syntaktischen Schlüsselwörter

$$\{\texttt{lambda},\ \texttt{quote},\ \texttt{if},\ \texttt{define},\ \texttt{set!},\ \texttt{set-car!},\ \texttt{set-cdr!},\ \texttt{begin}\}$$

vorkommen. Sei $\mathsf{Var}_0 := \{\texttt{cons}, \texttt{car}, \texttt{cdr}, \texttt{pair?}, \texttt{plus}, \texttt{times}\}$ die Menge der SCHEME-Variablen für vordefinierte (primitive) Prozeduren. Wir verwenden $x, x_1, \ldots$ als Mitteilungszeichen für SCHEME-Variablen. Die Menge Expr der SCHEME-*Ausdrücke* e wird durch folgende Regeln erzeugt:

(1) *Konstante*: $\mathsf{Konst} \subseteq \mathsf{Expr}$.
(2) *Variable*: $\mathsf{Var} \subseteq \mathsf{Expr}$.
(3) *Abstraktion*: $(\texttt{lambda}\ (x_1\ \ldots\ x_n)\ e)$.
(4) *Anwendung*: $(e\ e_1\ \ldots\ e_n)$.
(5) *Fallunterscheidung*: $(\texttt{if}\ e\ e_1\ e_2)$.
(6) *Definition*: $(\texttt{define}\ x\ e)$.
(7) *Zuweisung*: $(\texttt{set!}\ x\ e)$.
(8) *Veränderung von Daten*: $(\texttt{set-car!}\ e\ e_1)$, $(\texttt{set-cdr!}\ e\ e_1)$.
(9) *Block*: $(\texttt{begin}\ e_1\ \ldots\ e_n)$.

Jeder SCHEME-Ausdruck ist also ein endliches Datenobjekt, d.h. $\mathsf{Expr} \subseteq \mathsf{Data}$.

ÜBUNG 1.1. Man schreibe eine Prozedur, die ein Objekt daraufhin überprüft, ob es ein korrekter SCHEME-Ausdruck ist.

1.3. Werte, Rahmen und Umgebungen. 1. Sei

$$\begin{aligned} \mathsf{Pair} &:= \{\mathsf{pair}\} \times \mathbb{N}, \\ \mathsf{PrimProc} &:= \{\, [x] \mid x \in \mathsf{Var}_0 \,\}, \\ \mathsf{Proc} &:= \mathsf{Pair} \times \mathbb{N}^*. \end{aligned}$$

Pair ist die Menge der (Codes für) SCHEME-*Paare*, $\mathsf{PrimProc}$ die Menge der (Namen für) *vordefinierte* SCHEME-*Prozeduren* und Proc die Menge der (Codes für) *zusammengesetzte Prozeduren*, d.h. Werte von Lambda-Ausdrücken. Dabei kodiert in einem Wert $(w, \alpha) \in \mathsf{Proc}$ die linke Komponente w einen Lambda-Ausdruck e und die rechte Komponente α die für die freien Variablen von e gültige lokale Variablenumgebung (s.u.). Die Menge V der *Werte* ist definiert durch

$$\mathsf{V} := \mathsf{atom} \cup \mathsf{Pair} \cup \mathsf{PrimProc} \cup \mathsf{Proc}.$$

2. Ein *Rahmen* ist eine endliche Abbildung $F\colon \mathsf{dom}(F) \to \mathsf{V}$ mit $\mathsf{dom}(F) \subseteq \mathsf{Var}$. Frame sei die Menge der Rahmen.

3. Eine *Umgebung* $U = (U_v, U_p)$ besteht aus zwei partiellen Abbildungen

$$\begin{aligned} U_v &\colon \mathbb{N}^* \to^{\cup} \mathsf{Frame} \quad &\text{(Variablenumgebung)},\\ U_p &\colon \mathbb{N} \to^{\cup} \mathsf{V} \times \mathsf{V} \quad &\text{(Paarumgebung)}, \end{aligned}$$

wobei $\mathsf{dom}(U_v)$ ein endlicher Baum ist. Statt $U_v(\alpha)$ bzw. $U_p(n)$ schreiben wir meist kurz $U(\alpha)$ bzw. $U(n)$. Auch bei $U_v[\alpha \mapsto F]$ und $U_p[n \mapsto (w, v)]$ lassen wir die Indizes meist weg. Ist $\alpha = \langle n_1, \dots, n_k\rangle \in \mathsf{dom}(U)$, so ist

$$U_\alpha := U(\varepsilon)[U(\langle n_1\rangle)][U(\langle n_1, n_2\rangle)] \dots [U(\alpha)]$$

die *lokale Variablenumgebung* am Ort α. Eine Variablenumgebung kann man sich also als eine baumartig zusammenhängende Menge von lokalen Variablenumgebungen vorstellen. U_ε $(= U(\varepsilon))$ ist die „sichtbare" lokale Variablenumgebung. U_α mit nichtleerem α sind lokale Variablenumgebungen, in denen Teilberechnungen stattfinden. Wir verwenden die Abkürzung

$$U[\alpha, \vec{x} \mapsto \vec{w}] := U_v[\alpha \mapsto U(\alpha)[\vec{x} \mapsto \vec{w}]].$$

Ferner sei $\mathsf{parent}(x, \alpha, U)$ das längste Anfangsstück β von α derart daß $x \in \mathsf{dom}(U(\beta))$. Falls so ein β nicht existiert, ist $\mathsf{parent}(x, \alpha, U)$ undefiniert. Offensichtlich ist

$$U_\alpha(x) = U(\mathsf{parent}(x, \alpha, U))(x).$$

Mit Env bezeichnen wir die Menge aller Umgebungen.

4. Eine *Standardumgebung* ist eine Umgebung U, in der an der Wurzel die Variablen aus Var_0 durch die entsprechenden primitiven Prozeduren interpretiert sind. Es muß also gelten $\mathsf{Var}_0 \subseteq \mathsf{dom}(U_\varepsilon)$ und $U_\varepsilon(x) = [x]$ für alle $x \in \mathsf{Var}_0$.

1.4. Die Darstellung eines Scheme-Ausdrucks in einer Umgebung. In einer Umgebung lassen sich endliche Datenobjekte und damit SCHEME-Ausdrücke darstellen. Die Relation $w =_U d$ mit der Bedeutung „der Wert w stellt in der Umgebung U das endliche Datenobjekt d dar" ist definiert durch

(1) $a =_U a$, falls $a \in \mathsf{atom}$.
(2) Wenn $w = (\mathsf{pair}, n)$, $U_p(n) = (w_1, w_2)$ und $w_i =_U d_i$, dann gilt $w =_U (d_1 \ . \ d_2)$.

Da jeder SCHEME-Ausdruck ein endliches Datenobjekt ist, ist damit erklärt, wann ein Wert w in einer Umgebung U einen SCHEME-Ausdruck e darstellt.

Für spätere Zwecke definieren wir noch eine Relation $w =_U (w_1 \ \dots \ w_k)$ mit der Bedeutung „der Wert w stellt in der Umgebung U die Liste der Werte $w_1, \dots, w_k$ dar" durch

(1) $() =_U ()$.
(2) Wenn $w = (\mathsf{pair}, n) \in \mathsf{Pair}$, $U_p(n) = (w_1, v)$ und $v =_U (w_2 \ \dots \ w_k)$, dann gilt $w =_U (w_1 \ w_2 \ \dots \ w_k)$.

Offensichtlich gilt

$$w =_U (e_1\ \dots\ e_k) \Leftrightarrow$$
$$\exists w_1, \dots, w_k. [w =_U (w_1\ \dots\ w_k) \text{ und } w_i =_U e_i \text{ für } i = 1, \dots, k].$$

Hierbei ist $w =_U (e_1\ \dots\ e_k)$ eine Instanz von $w =_U d$, wobei d der Scheme-Ausdruck $(e_1\ \dots\ e_k)$ ist, während $w =_U (w_1\ \dots\ w_k)$ wie eben definiert ist.

1.5. Auswertung und Anwendung. Durch simultane Rekursion definieren wir partielle Funktionen

$$\mathsf{eval}\colon \mathsf{V} \times \mathbb{N}^* \times \mathsf{Env} \to^{\cup} \mathsf{V} \times \mathsf{Env},$$
$$\mathsf{apply}\colon \mathsf{V} \times \mathsf{V}^* \times \mathsf{Env} \to^{\cup} \mathsf{V} \times \mathsf{Env}.$$

Falls $w =_U e$, so ist $\mathsf{eval}(w, \alpha, U)$ der Wert des Ausdrucks e am Ort α in der Umgebung U, zusammen mit einer eventuell veränderten Umgebung U'. $\mathsf{apply}(w, (w_1, \dots, w_n), U)$ ist der Wert der Prozedur w an den Argumenten $w_1, \dots, w_n$ in der Umgebung U, wieder zusammen mit einer eventuell veränderten Umgebung U'. Falls w keine Prozedur ist, so ist das Ergebnis undefiniert.

Bemerkung. Man ist nur interessiert an der Auswertung von `Scheme`-Ausdrücken; es liegt deshalb die Frage nahe, warum man eval nicht auf der Menge Expr der `Scheme`-Ausdrücke statt auf der Menge V der Werte definiert. Der Grund ist, daß man dann etwa im Fall `quote` in Schwierigkeiten käme.

eval. Wir beginnen mit der Definition von eval.

$$\mathsf{eval}(\#t, \alpha, U) := (\#t, U),$$
$$\mathsf{eval}(\#f, \alpha, U) := (\#f, U),$$
$$\mathsf{eval}(n, \alpha, U) := (n, U),$$
$$\mathsf{eval}((\mathtt{quote}\ d), \alpha, U) := (d, U),$$

wobei letzteres als Abkürzung zu lesen ist für

$$\text{wenn } w =_U (\mathtt{quote}\ w_1) \text{ und } w_1 =_U d, \text{ dann } \mathsf{eval}(w, \alpha, U) := (w_1, U).$$

$$\mathsf{eval}(x, \alpha, U) := (U_\alpha(x), U),$$
$$\mathsf{eval}((\mathtt{lambda}\ (\vec{x})\ e), \alpha, U) := (((\mathtt{lambda}\ (\vec{x})\ e), \alpha), U),$$

wobei letzteres als Abkürzung zu lesen ist für

$$\text{wenn } w =_U (\mathtt{lambda}\ (\vec{x})\ e), \text{ dann } \mathsf{eval}(w, \alpha, U) := ((w, \alpha), U).$$

Die folgenden Gleichungen sind wie die Gleichung für `quote` zu lesen. Gilt $w =_U e$, so schreiben wir oft $\mathsf{eval}(e, \alpha, U)$ für $\mathsf{eval}(w, \alpha, U)$.

$$\mathsf{eval}((e\ e_1\ \dots\ e_n), \alpha, U) := \mathsf{apply}(w, \langle w_1, \dots, w_n\rangle, U'),$$

wobei $\mathsf{eval}(e, \alpha, U_n) = (w, U')$ mit $U_0 := U$ und $\mathsf{eval}(e_i, \alpha, U_{i-1}) = (w_i, U_i)$ für $i = 1, \dots, n$. Die Reihenfolge der Auswertung von links nach rechts wird nicht immer eingehalten. Viele SCHEME-Implementierungen werten Anwendungen von rechts nach links aus.

$$\mathsf{eval}((\texttt{if}\ e\ e_1\ e_2), \alpha, U) := \begin{cases} \mathsf{eval}(e_1, \alpha, U') & \text{falls } w \text{ definiert ist, aber} \neq \#f, \\ \mathsf{eval}(e_2, \alpha, U') & \text{falls } w \text{ definiert ist und} = \#f, \\ \text{undefiniert} & \text{sonst,} \end{cases}$$

wobei $\mathsf{eval}(e, \alpha, U) = (w, U')$.

$$\mathsf{eval}((\texttt{define}\ x\ e), \alpha, U) := (\texttt{unspecified}, U'[\alpha, x \mapsto w]),$$
$$\mathsf{eval}((\texttt{set!}\ x\ e), \alpha, U) := (\texttt{unspecified}, U'[\beta, x \mapsto w]),$$

wobei $\mathsf{eval}(e, \alpha, U) = (w, U')$ und $\beta = \mathsf{parent}(x, \alpha, U)$.

$$\mathsf{eval}((\texttt{set-car!}\ e\ e_1), \alpha, U) := (\texttt{unspecified}, U''[n \mapsto (w_1, v_2)]),$$
$$\mathsf{eval}((\texttt{set-cdr!}\ e\ e_1), \alpha, U) := (\texttt{unspecified}, U''[n \mapsto (v_1, w_1)]),$$

falls $\mathsf{eval}(e_1, \alpha, U) = (w_1, U')$, $\mathsf{eval}(e, \alpha, U') = ((\mathsf{pair}, n), U'')$ und $U''(n) = (v_1, v_2)$.

$$\mathsf{eval}((\texttt{begin}\ e_1 \ldots e_n), \alpha, U) := (w_n, U_n),$$

wobei $U_0 := U$ und $\mathsf{eval}(e_i, \alpha, U_{i-1}) = (w_i, U_i)$ für $i = 1, \ldots, n$.

apply. Wir definieren jetzt apply.

$$\mathsf{apply}(((\texttt{lambda}\ (\vec{x})\ e), \alpha), \vec{w}, U) := \mathsf{eval}(e, \alpha * \langle n \rangle, U[\alpha * \langle n \rangle, \vec{x} \mapsto \vec{w}]),$$

wobei n minimal ist mit $\alpha * \langle n \rangle \notin \mathsf{dom}(U_v)$. Gemeint ist dabei: Wenn $(\mathsf{pair}, m) =_U (\texttt{lambda}\ (\vec{x})\ e)$, und $w =_U e$, so $\mathsf{apply}(((\mathsf{pair}, m), \alpha), \vec{w}, U) := \mathsf{eval}(w, \alpha * \langle n \rangle, U[\alpha * \langle n \rangle, \vec{x} \mapsto \vec{w}])$.

$$\mathsf{apply}([\texttt{cons}], (w_1, w_2), U) := ((\mathsf{pair}, n), U[n \mapsto (w_1, w_2)]),$$

wobei n minimal ist mit $n \notin \mathsf{dom}(U_p)$.

$$\mathsf{apply}([\texttt{car}], ((\mathsf{pair}, n)), U) := (\pi_1(U_p(n)), U),$$
$$\mathsf{apply}([\texttt{cdr}], ((\mathsf{pair}, n)), U) := (\pi_2(U_p(n)), U),$$
$$\mathsf{apply}([\texttt{pair?}], (w), U) := \begin{cases} (\#t, U) & \text{falls } w \in \mathsf{Pair}, \\ (\#f, U) & \text{falls } w \notin \mathsf{Pair}, \end{cases}$$
$$\mathsf{apply}([\texttt{integer?}], (w), U) := \begin{cases} (\#t, U) & \text{falls } w \in \mathbb{N}, \\ (\#f, U) & \text{falls } w \notin \mathbb{N}, \end{cases}$$
$$\mathsf{apply}([\texttt{plus}], (n_1, \ldots, n_k), U) := (n_1 + \ldots + n_k, U),$$
$$\mathsf{apply}([\texttt{times}], (n_1, \ldots, n_k), U) := (n_1 * \ldots * n_k, U).$$

In allen anderen Fällen sind eval und apply undefiniert.

2. Korrektheit des Umgebungsmodells

Um eine Korrektheitsaussage über das Umgebungsmodell machen zu können, müssen wir uns über die intendierte Bedeutung eines SCHEME-Ausdrucks klar werden. Für Ausdrücke mit Seiteneffekt wie (`set!` x e) aber auch für viele rein funktionale Ausdrücke wie (`lambda` (x) (x x)) ist das recht schwierig. Wir beschränken uns daher auf ein Fragment von SCHEME, welches eine einfache (mengentheoretische) Interpretation besitzt. Wir betrachten nur solche Ausdrücke, die kein `set!`, `set-car!` oder `set-cdr!`

enthalten und deren Variablen in konsistenter Weise Typen, d.h. Funktionalitäten, zugeordnet werden können. Da unser Modell keine partiellen Funktionen enthalten wird, müssen partielle Prozeduren wie `car` und `cdr` ersetzt werden durch totale Versionen `safecar` und `safecdr`, die zunächst abfragen, ob ihr Argument ein Paar ist. Ferner erlauben wir die Paarbildung mit `cons` nur für Objekte des „Grundtyps“ o.

2.1. Getypte Ausdrücke und ihre mengentheoretische Semantik. Die Menge der *Typen* ist erklärt durch

(1) o ist ein Typ.
(2) Sind $\rho_1, \dots, \rho_n, \sigma$ Typen, so ist auch $(\rho_1, \dots, \rho_n) \to \sigma$ ein Typ.

Statt $(\rho) \to \sigma$ schreiben wir auch $\rho \to \sigma$. Ist x eine Variable und ist ρ ein Typ, so ist x^ρ eine *getypte Variable*. Wir definieren induktiv eine Menge von Ausdrücken und für jeden dieser Ausdrücke seinen Typ.

(1) `#t`, `#f`, sowie (`quote` d) für $d \in \mathsf{Data}$ sind Ausdrücke vom Typ o.
(2) Jede getypte Variable x^ρ ist ein Ausdruck vom Typ ρ.
(3) Sind $x_1^{\rho_1}, \dots, x_n^{\rho_n}$ verschiedene getypte Variablen und ist e ein Ausdruck vom Typ σ, so ist (`lambda` ($x_1^{\rho_1}$ ... $x_n^{\rho_n}$) e) ein Ausdruck vom Typ $(\rho_1, \dots, \rho_n) \to \sigma$.
(4) Ist e ein Ausdruck vom Typ $(\rho_1, \dots, \rho_n) \to \sigma$ und sind $e_1, \dots, e_n$ Ausdrücke vom Typ $\rho_1, \dots, \rho_n$ respektive, so ist (e e_1 ... e_n) ein Ausdruck vom Typ σ.
(5) Ist e ein Ausdruck vom Typ o und sind e_1, e_2 Ausdrücke vom Typ ρ, so ist (`if` e e_1 e_2) ein Ausdruck vom Typ ρ.

Wir betrachten nur solche Ausdrücke, in denen jede vorkommende Variable genau einen Typ hat. Wir schreiben häufig e^ρ statt „e ist ein Ausdruck vom Typ ρ“.

Für jeden Typ ρ definieren wir eine Menge M_ρ.

$$M_o := \mathsf{Data}, \quad M_{\vec{\rho}\to\sigma} := M_\sigma^{M_{\vec{\rho}}},$$

wobei $\vec{\rho} = (\rho_1, \dots, \rho_n)$ und $M_{\vec{\rho}} := M_{\rho_1} \times \dots \times M_{\rho_n}$. Eine *Bewertung* ist eine Abbildung η, die endlich vielen getypten Variablen $x_1^{\rho_1}, \dots, x_n^{\rho_n}$ Objekte $\eta(x_i^{\rho_i}) \in M_{\rho_i}$ zuordnet. Für jeden Ausdruck e vom Typ ρ und jede Bewertung η mit $\mathsf{FV}(e) \subseteq \mathsf{dom}(\eta)$ definieren wir seine *Interpretation* $[\![e]\!]_\eta \in M_\rho$.

1. $[\![\#t]\!]_\eta := \#t$, $[\![\#f]\!]_\eta := \#f$, $[\![(\texttt{quote}\ d)]\!]_\eta := d$.
2. $[\![x^\rho]\!]_\eta := \eta(x^\rho)$.
3. $[\![(\texttt{lambda}\ (\vec{x}^{\vec{\rho}})\ e^\sigma)]\!]_\eta := f \in M_{\vec{\rho}\to\sigma}$ mit $f(\vec{a}) := [\![e]\!]_{\eta[\vec{x}^{\vec{\rho}} \mapsto \vec{a}]}$ für alle $\vec{a} \in M_{\vec{\rho}}$.
4. $[\![(e^{\vec{\rho}\to\sigma}\ e_1^{\rho_1}\ \dots\ e_n^{\rho_n})]\!]_\eta := [\![e]\!]_\eta([\![e_1]\!]_\eta, \dots, [\![e_n]\!]_\eta)$.
5. $[\![(\texttt{if}\ e^o\ e_1^\rho\ e_2^\rho)]\!]_\eta := \begin{cases} [\![e_1]\!]_\eta & \text{falls } [\![e]\!]_\eta \neq \#f, \\ [\![e_2]\!]_\eta & \text{sonst.} \end{cases}$

2.2. Der Korrektheitsbeweis. Für jeden getypten Ausdruck e sei $\mathsf{del}(e)$ der SCHEME-Ausdruck, der durch Streichen aller Typen entsteht. Ferner definieren wir eine Bewertung η_0 durch

$$\mathsf{dom}(\eta_0) := \{\texttt{cons}^{(o,o)\to o}, \texttt{safecar}^{o\to o}, \texttt{safecdr}^{o\to o}\},$$

wobei

$$\eta_0(\mathtt{cons}^{(o,o)\to o})(d_1,d_2) := (d_1\ .\ d_2),$$

$$\eta_0(\mathtt{safecar}^{o\to o})(d) := \begin{cases} d_1 & \text{falls } d = (d_1\ .\ d_2),\\ \#f & \text{sonst,}\end{cases}$$

$$\eta_0(\mathtt{safecdr}^{o\to o})(d) := \begin{cases} d_2 & \text{falls } d = (d_1\ .\ d_2),\\ \#f & \text{sonst.}\end{cases}$$

Der folgende Korrektheitssatz bezieht sich auf *erweiterte Standardumgebungen*, d.h. Standardumgebungen (siehe Abschnitt 1.3), für die zusätzlich $\mathtt{safecar}, \mathtt{safecdr} \in \mathsf{dom}(U_\varepsilon)$ und für $[\mathtt{safecar}]_U := U_\varepsilon(\mathtt{safecar})$ und $[\mathtt{safecdr}]_U := U_\varepsilon(\mathtt{safecdr})$ gilt

$$\mathsf{apply}([\mathtt{safecar}]_U, w, U) =$$
$$\begin{cases} (w_1, U) & \text{falls } w \in \mathsf{Pair}, \text{ etwa } w = (\mathtt{pair}, n) \text{ und } U_p(n) = (w_1, w_2),\\ \#f & \text{falls } w \notin \mathsf{Pair},\end{cases}$$

$$\mathsf{apply}([\mathtt{safecdr}]_U, w, U) =$$
$$\begin{cases} (w_2, U) & \text{falls } w \in \mathsf{Pair}, \text{ etwa } w = (\mathtt{pair}, n) \text{ und } U_p(n) = (w_1, w_2),\\ \#f & \text{falls } w \notin \mathsf{Pair}.\end{cases}$$

Offensichtlich kann man sich durch Auswertung von

```
(define safecar (lambda (x) (if (pair? x) (car x) #f)))
(define safecdr (lambda (x) (if (pair? x) (cdr x) #f)))
```

aus einer Standardumgebung eine erweiterte Standardumgebung verschaffen.

Für jeden Ausdruck e vom Typ o mit $\mathsf{FV}(e) \subseteq \mathsf{dom}(\eta_0)$, jede erweiterte Standardumgebung U und jedes $w \in \mathsf{V}$ mit $w =_U \mathsf{del}(e)$ wollen wir zeigen

$$\mathsf{eval}(w, \varepsilon, U) =_U [\![e]\!]_{\eta_0}.$$

Zum Beweis verwenden wir eine Version der auf TAIT zurückgehenden Technik der *Berechenbarkeitsprädikate*: Für Typen ρ, SCHEME-Werte $w \in \mathsf{V}$, Umgebungen U und Objekte $a \in M_\rho$ definieren wir eine Relation

$$(w, U) \sim_\rho a$$

(lies „(w, U) repräsentiert a") durch

$$(w, U) \sim_o d :\equiv w =_U d,$$
$$(w, U) \sim_{\vec\rho\to\sigma} f :\equiv \forall \vec w, \vec a, U' \supseteq U. (w_1, U') \sim_{\rho_1} a_1, \ldots, (w_n, U') \sim_{\rho_n} a_n \to$$
$$\mathsf{apply}(w, \vec w, U') \sim_\sigma f(\vec a).$$

In der zweiten Äquivalenz wird implizit verlangt, daß $\mathsf{apply}(w, \vec w, U')$ definiert ist.

Wir definieren außerdem

$$U \sim_\alpha \eta \quad :\Longleftrightarrow \quad \forall x^\rho \in \mathsf{dom}(\eta). (U_\alpha(x), U) \sim_\rho \eta(x^\rho).$$

LEMMA 2.1. *(1) Wenn $(w, U) \sim_\rho a$ und $U \subseteq U'$, so $(w, U') \sim_\rho a$.*

(2) *Wenn* $U \sim_\alpha \eta$ *und* $U \subseteq U'$, *so gilt* $U' \sim_\alpha \eta$.
(3) *Wenn* $w =_U \mathsf{del}(e)$ *und* $\mathsf{eval}(w, \alpha, U) = (w', U')$, *so gilt* $U \subseteq U'$.
(4) *Es ist* $U \sim_\varepsilon \eta_0$ *für jede erweiterte Standardumgebung* U.

Beweis. Übung. □

Theorem 2.2 (Korrektheit). *Sei* e *ein Ausdruck vom Typ* ρ, η *eine Bewertung mit* $\mathsf{FV}(e) \subseteq \mathsf{dom}(\eta)$, U *eine Umgebung,* $\alpha \in \mathsf{dom}(U)$, $w =_U \mathsf{del}(e)$ *und es gelte* $U \sim_\alpha \eta$. *Dann ist* $\mathsf{eval}(w, \alpha, U)$ *definiert und es gilt*

$$\mathsf{eval}(w, \alpha, U) \sim_\rho [\![e]\!]_\eta.$$

Beweis. Wir verwenden eine Induktion nach e. Gelte $U \sim_\alpha \eta$.

1. $\mathsf{eval}(\#t, \alpha, U) = (\#t, U) \sim_o \#t = [\![\#t]\!]$. Für die Fälle $\#f$ und () schließt man analog. Im Fall $\mathsf{eval}((\texttt{quote}\ d), \alpha, U) = (d, U)$, also genauer $\mathsf{eval}(w, \alpha, U) = (w_1, U)$ mit $w =_U (\texttt{quote}\ w_1)$ und $w_1 =_U d$ haben wir

$$\mathsf{eval}(w, \alpha, U) = (w_1, U) \sim_o d = [\![(\texttt{quote}\ d)]\!].$$

2. $\mathsf{eval}(x, \alpha, U) = (U_\alpha(x), U) \sim_\rho \eta(x^\rho) = [\![x^\rho]\!]_\eta$.

3. Gelte oBdA $\mathsf{dom}(\eta) = \mathsf{FV}((\texttt{lambda}\ (\vec{x}^{\vec{\rho}})\ e^\sigma))$. Ferner sei $(\mathsf{pair}, n) =_U (\texttt{lambda}\ (\vec{x})\ \mathsf{del}(e))$. Dann ist

$$\mathsf{eval}((\mathsf{pair}, n), \alpha, U) = (((\mathsf{pair}, n), \alpha), U).$$

Sei $f := [\![(\texttt{lambda}\ (\vec{x}^{\vec{\rho}})\ e^\sigma)]\!]_\eta$. Zu zeigen ist

$$(((\mathsf{pair}, n), \alpha), U) \sim_{\vec{\rho} \to \sigma} f.$$

Gelte also $U \subseteq U'$ und $(w_i, U') \sim_{\rho_i} a_i$ für $i = 1, \dots, n$. Zu zeigen ist

$$\mathsf{apply}(((\mathsf{pair}, n), \alpha), \vec{w}, U') \sim_\sigma f(\vec{a}).$$

Sei $w =_U \mathsf{del}(e)$. Dann gilt

$$\mathsf{apply}(((\mathsf{pair}, n), \alpha), \vec{w}, U') = \texttt{eval}(w, \alpha * \langle m \rangle, U'')$$

mit $U'' := U'[\alpha * \langle m \rangle, \vec{x} \mapsto \vec{w}]$, wobei $\alpha * \langle m \rangle \notin \mathsf{dom}(U_v)$. Also gilt $U \subseteq U' \subseteq U''$. Mit Lemma 2.1(1,2) und wegen $\mathsf{dom}(\eta) = \mathsf{FV}((\texttt{lambda}\ (\vec{x}^{\vec{\rho}})\ e))$ folgt $(w_i, U'') \sim_{\rho_i} a_i$ und $U'' \sim_{\alpha * \langle m \rangle} \eta$. Folglich ist $U'' \sim_{\alpha * \langle m \rangle} \eta[\vec{x}^{\vec{\rho}} \mapsto \vec{a}]$. Nach Induktionsvoraussetzung ist $\texttt{eval}(w, \alpha * \langle m \rangle, U'')$ definiert und es gilt

$$\texttt{eval}(w, \alpha * \langle m \rangle, U'') \sim_\sigma [\![e]\!]_{\eta[\vec{x}^{\vec{\rho}} \mapsto \vec{a}]} = f(\vec{a}).$$

4. Gelte $v =_U \mathsf{del}(e^{\vec{\rho} \to \sigma}\ e_1^{\rho_1} \dots e_n^{\rho_n})$. Dann gibt es $w, \vec{w}$ mit $w =_U \mathsf{del}(e)$, $w_i =_U \mathsf{del}(e_i)$ für $i = 1, \dots, n$ und es gilt $\mathsf{eval}(v, \alpha, U) = \mathsf{apply}(w', \vec{w}', U')$, wobei $U_0 := U$, $\mathsf{eval}(w_i, \alpha, U_{i-1}) = (w_i', U_i)$ für $i = 1, \dots, n$ und $\mathsf{eval}(w, \alpha, U_n) = (w', U')$. Ferner gilt $[\![(e\ e_1 \dots e_n)]\!]_\eta = f(\vec{a})$ mit $f := [\![e]\!]_\eta$ und $a_i := [\![e_i]\!]_\eta$. Zu zeigen ist

$$\mathsf{apply}(w', \vec{w}', U') \sim_\sigma f(\vec{a}).$$

Nach Lemma 2.1(2,3) gilt $U_i \sim_\alpha \eta$, $U' \sim_\alpha \eta$ und $U \subseteq U_1 \subseteq \dots \subseteq U_n \subseteq U'$. Es folgt $(w_i', U_i) \sim_{\rho_i} a_i$ und $(w', U') \sim_{\vec{\rho} \to \sigma} f$ nach Induktionsvoraussetzung. Mit Lemma 2.1(1) folgt $(w_i', U') \sim_{\rho_i} a_i$, also gilt $\mathsf{apply}(w', \vec{w}', U') \sim_\sigma f(\vec{a})$ nach Definition der Relation $\sim_{\vec{\rho} \to \sigma}$.

5. Gelte $v =_U \mathsf{del}(\texttt{if}\ e^o\ e_1^\rho\ e_2^\rho)$, also $v =_U (\texttt{if}\ w\ w_1\ w_2)$ mit $w =_U \mathsf{del}(e)$ und $w_i =_U \mathsf{del}(e_i)$. Nach Induktionsvoraussetzung ist $\mathsf{eval}(e, \alpha, U) = (w', U')$ definiert mit $w' =_{U'} [\![e]\!]_\eta$. Nach Lemma 2.1(2,3) gilt $U \subseteq U'$ und $U' \sim_\alpha \eta$. Folglich sind wieder nach Induktionsvoraussetzung auch $(w_i', U_i') :=$

$\mathsf{eval}(e_i, \alpha, U')$ definiert und es gilt $(w_i', U_i') \sim_\rho [\![e_i]\!]_\eta$. *Fall* $w' \neq \#f$. Dann gilt $[\![e]\!]_\eta \neq \#f$ und folglich

$$\mathsf{eval}(v, \alpha, U) = (w_1', U_1') \sim_\rho [\![e_1]\!]_\eta = [\![(\texttt{if}\ e\ e_1\ e_2)]\!]_\eta.$$

Fall $w' = \#f$. Analog. □

Mit $\rho = o$ und $\eta = \eta_0$ ergibt sich jetzt mit Hilfe von Lemma 2.1(4):

KOROLLAR 2.3. *Sei e ein Ausdruck vom Typ o mit* $\mathsf{FV}(e) \subseteq \mathsf{dom}(\eta_0)$, *U eine erweiterte Standardumgebung und* $w \in \mathsf{V}$ *mit* $w =_U \mathsf{del}(e)$. *Dann gilt*

$$\mathsf{eval}(w, \varepsilon, U) =_U [\![e]\!]_{\eta_0}.$$

3. Implementierung des Interpreters

Die aktuelle Umgebung $U = (U_v, U_p)$ wird durch zwei lange Vektoren (arrays) `var-env` und `pair-env` implementiert. Die baumartige Struktur der Variablenumgebung wird dadurch wiedergegeben, daß jeder Eintrag in `var-env` neben einem Rahmen noch den Index des darüberliegenden Rahmens enthält. Die Umgebungsvektoren `var-env` und `pair-env` werden den Prozeduren `eval` und `apply` nicht – wie in unserem mathematischen Modell – als Parameter übergeben. Vielmehr werden sie als globale Variablen gehalten und mittels Seiteneffekt (`vector-set!`) verändert.

Diese Darstellungsweise hat mehrere Vorteile. Zunächst entspricht sie der tatsächlichen Struktur eines Speichers im Rechner. Bei einer effizienteren Implementierung (etwa in der Programmsprache C) würde man ähnlich vorgehen. Weiter vermeidet man so unnötiges Kopieren von eventuell langen Ausdrücken.

Wir verwenden also `var-env` und `pair-env` als globale Variablen für genügend lange Vektoren.

```
(define length-of-var-env 1000)
(define length-of-pair-env 1000)

(define var-env (make-vector length-of-var-env))
(define pair-env (make-vector length-of-pair-env))
```

Ferner verwenden wir als globale Variable

```
(define largest-used-var-env-index -1)
(define largest-used-pair-env-index -1)
```

Ein typischer Eintrag im Vektor `var-env` hat die Form

$$(((x_1\ v_1)\ (x_2\ v_2)\ \ldots)\ k).$$

Hier ist k ein Zeiger auf den darüberliegenden Rahmen.

Die Einträge in `pair-env` sind Paare von Werten (letztere implementiert durch Zweier- oder Dreierlisten). Ein *Wert* hat stets die Form (`key` ...), wobei das Schlüsselwort `key` angibt, um welche Art eines Wertes es sich handelt.

(`true` $\#t$)

(`false` $\#f$)

(`nil` ())

(`number` n)

(`symbol` x)

(`pair` n)

(`proc` (`pair` n) k)

(`prim-proc` x) mit $x \in \mathsf{Var}_0 := \{\texttt{cons}, \texttt{car}, \texttt{cdr}, \texttt{pair?}, \texttt{plus}, \texttt{times}\}$

Bei (`proc` (`pair` n) k) beachte man, daß n ein Index eines Paares ist, der in der aktuellen Umgebung einen `lambda`-Ausdruck darstellt, und k ein Index eines Rahmens ist. Dieser Rahmen oder genauer seine Einordnung in dem Vektor `var-env` kodiert die lokale Umgebung, die beim Auswerten des `lambda`-Ausdrucks in Kraft war. `cons`, `car`, `cdr`, `pair?`, `plus` und `times` signalisieren, daß hier die entsprechenden primitiven Prozeduren als Werte gemeint sind.

```
(define (keyword v) (car v))
```

Zur besseren Lesbarkeit verwenden wir die Konvention, daß im allgemeinen mit w Werte bezeichnet werden, die SCHEME-Ausdrücke darstellen. Werte, für die dies nicht gemeint ist, werden meist mit v bezeichnet.

3.1. Die Variablenumgebung. Wir benötigen Konstruktor- und Selektorfunktionen für Umgebungen.

```
(define (make-frame vars vs k) (list (map list vars vs) k))
(define (frame-to-bindings frame) (car frame))
(define (frame-to-parent frame) (cadr frame))
```

Wir benötigen weiter folgende Hilfsfunktionen für Umgebungen.

```
(define (insert-frame frame)
  (set! largest-used-var-env-index
        (+ largest-used-var-env-index 1))
  (if (>= largest-used-var-env-index length-of-var-env)
      (error 'var-env "Stack overflow")
      (begin
        (vector-set! var-env largest-used-var-env-index frame)
        largest-used-var-env-index)))

(define (index-to-frame k) (vector-ref var-env k))

(define (var-env-lookup x k)
  (let* ((frame (index-to-frame k))
         (bindings (frame-to-bindings frame))
         (info (assq x bindings)))
    (if info
```

```
        (cadr info)
        (if (= 0 k)
            (error 'var-env-lookup "Unbound variable ~s" x)
            (var-env-lookup x (frame-to-parent frame))))))
```

Bei der Hinzunahme einer Variablenbelegung zum aktuellen Frame überprüfen wir, ob sie bereits belegt ist und ändern in diesem Fall lediglich den Wert.

```
(define (var-env-add! x v k)
  (let* ((frame (index-to-frame k))
         (bindings (frame-to-bindings frame))
         (info (assq x bindings)))
    (if info
        (set-car! (cdr info) v) ;Veraenderung des Wertes
        (set-car! frame (cons (list x v) bindings)))))
```

Die Veränderung von Variablenwerten durch `var-env-mutate!` geschieht entsprechend.

```
(define (var-env-mutate! x v k)
  (let* ((frame (index-to-frame k))
         (bindings (frame-to-bindings frame))
         (info (assq x bindings)))
    (if info
        (set-car! (cdr info) v) ;Veraenderung des Wertes
        (if (= 0 k)
            (error 'var-env-mutate! "Unbound variable ~s" x)
            (var-env-mutate! x v (frame-to-parent frame))))))
```

Anzeigefunktion:

```
(define (show-frame k)
  (if (<= k largest-used-var-env-index)
      (let* ((frame (index-to-frame k))
             (bindings (frame-to-bindings frame))
             (parent (frame-to-parent frame)))
        (display (string-append "frame number "
                                (number->string k)
                                " has parent-frame number "
                                (number->string parent)
                                " and bindings"))
        (newline)
        (do ((rest bindings (cdr rest)))
            ((null? rest))
          (display (caar rest))
          (display " -> ")
```

```
          (display (cadar rest))
          (newline)))))

(define (show-var-env from-index to-index)
  (let ((max-index (max to-index largest-used-var-env-index)))
    (do ((i from-index (+ 1 i)))
        ((> i max-index))
      (show-frame i))))
```

3.2. Die Paarumgebung. Wir benötigen ferner Konstruktor- und Selektorfunktionen für Objekte.

```
(define (make-pair v1 v2)
  (set! largest-used-pair-env-index
        (+ largest-used-pair-env-index 1))
  (if (>= largest-used-pair-env-index length-of-pair-env)
      (error 'pair-env "Stack overflow")
      (begin
        (vector-set! pair-env
                     largest-used-pair-env-index
                     (list v1 v2))
        (list 'pair largest-used-pair-env-index))))

(define (pair-to-car v) (car (vector-ref pair-env (cadr v))))
(define (pair-to-cdr v) (cadr (vector-ref pair-env (cadr v))))
```

Wir benötigen weiter folgende Hilfsfunktionen für Paare.

```
(define (pair-env-mutate-car! pair-value value)
  (let* ((pair-index (cadr pair-value))
         (cdr-pair (cadr (vector-ref pair-env pair-index))))
    (vector-set! pair-env pair-index (list value cdr-pair))))

(define (pair-env-mutate-cdr! pair-value value)
  (let* ((pair-index (cadr pair-value))
         (car-pair (car (vector-ref pair-env pair-index))))
    (vector-set! pair-env pair-index (list car-pair value))))
```

Die Anzeigefunktion `(show-pair-env n)` zeigt alle Paare, die seit der letzten Anzeige hinzugekommen sind.

```
(define (show-pair index)
  (display index)
  (display ": (")
  (let ((pair (vector-ref pair-env index)))
    (display (car pair))
    (display " . ")
```

```
    (display (cadr pair))
    (display ")")))

(define last-displayed-pair 0)

(define (show-pair-env to-index)
  (do ((i last-displayed-pair (+ i 1)))
      ((or (> i to-index)
           (null? (vector-ref pair-env i)))
       (set! last-displayed-pair i))
    (show-pair i)
    (newline)))

(define (show-last-pairs)
  (show-pair-env largest-used-pair-env-index))
```

`show-all-pairs` zeigt die gesamte Paar-Umgebung.

```
(define (show-all-pairs)
  (set! last-displayed-pair 0)
  (show-last-pairs))
```

3.3. Hilfsfunktionen. Als nächstes definieren wir einige Hilfsfunktionen zur Umwandlung von benutzerdefinierten Prozeduren in Werte, und für das Bearbeiten solcher Werte. Verwendet wird eine Prozedur `val-to-list`, die den Wert `v` einer Liste in eine Liste von Werten umwandelt.

```
(define (val-to-list v)
  (do ((v0 v (pair-to-cdr v0))
       (l '() (cons (pair-to-car v0) l)))
      ((eq? 'nil (keyword v0)) (reverse l))))

(define (make-proc w k) (list 'proc w k))
(define (proc-to-lambda-expr proc) (cadr proc))
(define (proc-to-frame-index proc) (caddr proc))
(define (lambda-expr-to-vars w)
  (map cadr (val-to-list (cadr (val-to-list w)))))
(define (lambda-expr-to-body w) (caddr (val-to-list w)))
```

Wir teilen die primitiven Prozeduren in solche ein, die auf Atomen arbeiten, und solche, die allgemeinere Werte erwarten. Wir verwenden dazu eine Assoziationsliste `prim-procs-on-atoms`, die die verwendeten primitiven Prozeduren auf Atomen zusammen mit ihren Namen auflistet. Dies erleichtert das Hinzufügen weiterer solcher primitiver Prozeduren.

```
(define prim-procs-on-atoms (list (list '+ +)
                                  (list '- -)
                                  (list '* *)
```

```
                                 (list '/ /)
                                 (list 'max max)
                                 (list '= =)
                                 (list '<= <=)
                                 (list '>= >=)
                                 (list '< <)
                                 (list '> >)))
```

Weiter definieren wir `prim-procs-on-values`. Ferner verwenden wir gewisse report-Funktionen; sie erstatten nur dann Bericht, wenn `trace?` auf wahr gesetzt ist.

```
(define trace? #f)

(define (report-pairs)
  (if trace? (show-last-pairs)))

(define (report . infos)
  (if trace? (for-each display infos)))

(define (report-and-newline . infos)
  (apply report infos)
  (if trace? (newline)))

(define true-value '(true #t))
(define false-value '(false #f))
(define nil-value '(nil ()))

(define prim-procs-on-values
  (list (list 'cons
              (lambda (value1 value2)
                (let ((result (make-pair value1 value2)))
                  (report-and-newline " cons adds new pair: ")
                  (report-pairs)
                  result)))
        (list 'car pair-to-car)
        (list 'cdr pair-to-cdr)
        (list 'set-car!
              (lambda (pair-value value)
                (pair-env-mutate-car! pair-value value)
                (report-and-newline
                 " set-car! changes pair-env: ")
                (if trace? (show-pair (cadr pair-value)))
                pair-value))
        (list 'set-cdr!
              (lambda (pair-value value)
                (pair-env-mutate-cdr! pair-value value)
                (report-and-newline
```

```
                 " set-cdr! changes pair-env: ")
                (if trace? (show-pair (cadr pair-value)))
                pair-value))
        (list 'pair?
              (lambda (value)
                (if (eq? 'pair (keyword value))
                    true-value
                    false-value)))
        (list 'null?
              (lambda (value)
                (if (eq? 'nil (keyword value))
                    true-value
                    false-value)))
        (list 'number?
              (lambda (value)
                (if (eq? 'number (keyword value))
                    true-value
                    false-value)))
        (list 'ev (lambda (value) (eval value 0)))))
```

Zu Beginn benötigen wir eine Hilfsprozedur `clear`, die die Vektoren `var-env` und `pair-env` geeignet initialisiert.

```
(define (clear)
  (set! largest-used-var-env-index -1)
  (set! largest-used-pair-env-index -1)
  (vector-fill! var-env '())
  (vector-fill! pair-env '())
  (insert-frame
    (let* ((x (append (map car prim-procs-on-atoms)
                      (map car prim-procs-on-values)))
           (y (map (lambda (p) (list 'prim-proc p)) x)))
      (make-frame x y 0))))
```

3.4. Die eval-Funktion. Wir können jetzt `(eval w k)` definieren. Dazu legen wir die syntaktischen Schlüsselwörter fest.

```
(define syntactic-keywords
  '(lambda quote if define set! begin))
```

Falls `w` boolesches Objekt oder eine Zahl ist, wird `w` zurückgegeben. Falls `w` ein Symbol ist, wird die Bedeutung an der Stelle `k` in `var-env` nachgesehen. Falls `w` eine spezielle Form ist, also mit `lambda`, `quote`, `if`, `define` oder `set!` beginnt, so ist `w` ein Paar mit dem Schlüsselwort als `w0`, und es müssen gewisse Bedingungen an `w1` erfüllt sein. Die Definition von `(eval w k)` ist dann klar. Falls `w` ein Anwendungsausdruck ist, ist `w` ein Paar beginnend mit `w0`, und es folgen Null oder mehr Argumente `ws`. Man berechnet dann die

Werte `v0,vs` von `w0,ws` an der Stelle `k` in `var-env` und ruft `(app v0 vs)` auf. Zu beachten ist, daß `v0` eine Prozedur sein muß.

```
(define (eval w k)
  (let ((key (keyword w)))
    (cond
     ((eq? 'true key) w)
     ((eq? 'false key) w)
     ((eq? 'number key) w)
     ((eq? 'symbol key) (var-env-lookup (cadr w) k))
     (else
      (let* ((ws (val-to-list w))
             (w0 (car ws))
             (key0 (keyword w0)))
        (if (and (eq? 'symbol key0)
                 (memq (cadr w0) syntactic-keywords))
            (let ((x (cadr w0)))
              (cond
               ((eq? 'lambda x) (make-proc w k))
               ((eq? 'quote x) (cadr ws))
               ((eq? 'if x)
                (let* ((w1 (cadr ws))
                       (w2 (caddr ws))
                       (w3 (cadddr ws))
                       (key1 (keyword (eval w1 k))))
                  (if (eq? 'false key1)
                      (eval w3 k)
                      (eval w2 k))))
               ((eq? 'define x)
                (let ((y (cadr (cadr ws)))
                      (w1 (caddr ws)))
                  (var-env-add! y (eval w1 k) k)
                  (list 'symbol y)))
               ((eq? 'set! x)
                (let ((y (cadr (cadr ws)))
                      (w1 (caddr ws)))
                  (var-env-mutate! y (eval w1 k) k)
                  (list 'symbol y)))
               ((eq? 'set-car! x)
                (let ((w (cadr ws))
                      (w1 (caddr ws)))
                  (pair-env-mutate-car! w w1)
                  w1))
               ((eq? 'set-cdr! x)
                (let ((w (cadr ws))
                      (w1 (caddr ws)))
                  (pair-env-mutate-cdr! w w1)
                  w1))
```

```
          ((eq? 'begin x)
           (do ((l (cdr ws) (cdr l))
                (res '() (eval (car l) k)))
               ((null? l) res)))))
       ;sonst Anwendungsterm
       (let* ((vs (do ((l (cdr ws) (cdr l))
                       (res '() (cons (eval (car l) k)
                                      res)))
                      ((null? l) (reverse res))))
              (v0 (eval w0 k)))
         (app v0 vs))))))))
```

3.5. Die app-Funktion. Zur Definition von `(app v vs)` verwenden wir eine Fallunterscheidung, ob `v` eine benutzerdefinierte oder primitive Prozedur ist. Im ersten Fall greift man auf `eval` zurück.

```
(define (atom-to-atom-val a)
  (cond ((boolean? a) (if a true-value false-value))
        ((null? a) nil-value)
        ((and (integer? a) (not (negative? a)))
         (list 'number a))
        ((symbol? a) (list 'symbol a))
        (else (error 'atom-to-atom-val "Unknown atom ~s" a))))

(define (app v vs)
  (let ((key (keyword v)))
    (cond
      ((eq? 'proc key)
       (let* ((k (proc-to-frame-index v))
              (w (proc-to-lambda-expr v))
              (vars (lambda-expr-to-vars w))
              (w1 (lambda-expr-to-body w)))
         (eval w1 (insert-frame (make-frame vars vs k)))))
      ((eq? 'prim-proc key)
       (let* ((symbol (cadr v))
              (info (assq symbol prim-procs-on-atoms)))
         (if info
             (atom-to-atom-val
               (apply (cadr info) (map cadr vs)))
             ;sonst primitive Prozedur auf Werten
             (apply (cadr (assq symbol prim-procs-on-values))
                    vs)))))))
```

ÜBUNG 3.1. Schreibe `user-display`. Eingabe: Ein Wert $\in \mathsf{V}$ (genauer: seine Implementierung). Ausgabe: Das entsprechende SCHEME-Objekt in seiner Paar-Struktur (die Prozedur-Werte an den Blättern sollen unverändert ausgegeben werden).

ÜBUNG 3.2. Schreibe eine Prozedur `user-read`, die beim Einlesen eines SCHEME-Ausdrucks den Vektor `pair-env` entsprechend verändert und einen Wert ∈ V zurückgibt.

LÖSUNG.

```
(define (user-display v)
  (let ((key (keyword v)))
    (cond ((eq? key 'pair)
           (cons (user-display (pair-to-car v))
                 (user-display (pair-to-cdr v))))
          ((eq? key 'true) '#t)
          ((eq? key 'false) '#f)
          ((eq? key 'nil) '())
          ((eq? key 'number) (cadr v))
          ((eq? key 'symbol) (cadr v))
          (else v))))

(define (user-read e)
  (cond ((pair? e) (let ((w1 (user-read (car e)))
                         (w2 (user-read (cdr e))))
                     (make-pair w1 w2)))
        ((eq? e '#t) true-value)
        ((eq? e '#f) false-value)
        ((null? e) nil-value)
        ((and (integer? e) (not (negative? e)))
         (list 'number e))
        ((symbol? e) (list 'symbol e))
        (else (error 'user-read
                     "~s is not a finite data object" e))))
```

4. Beispiele

Schließlich noch einige Probeläufe des Interpreters:

```
(clear)
(define (evl e) (user-display (eval (user-read e) 0)))
```

4.1. Paare. Wir beginnen mit Beispielen zur speziellen Rolle von Paaren in SCHEME:

```
(evl '(define x (quote (0 1))))
(evl '(define y x))
(evl '(set-car! (cdr x) 2))
(evl 'y) ==> (0 2)
(evl '(set! x (quote (0 3))))
(evl 'y) ==> (0 2)
```

Um zu verstehen, was hier vorgeht, sehen wir uns den Ablauf der Rechnung genauer an.

```
(set! trace? #t)
(evl '(define x (quote (0 1))))
(evl '(define y x))
(evl '(set-car! (cdr x) 2))

(show-frame 0) ==>
frame number 0 has parent-frame number 0 and bindings
y -> (pair 1)
x -> (pair 1)

(show-pair 1) ==> 1: ((number 0) . (pair 0))
(show-pair 0) ==> 0: ((number 2) . (nil ()))

(evl 'y) ==> (0 2)

(evl '(set! x (quote (0 3))))

(show-frame 0) ==>
frame number 0 has parent-frame number 0 and bindings
y -> (pair 1)
x -> (pair 16)

(show-pair 16) ==> 16: ((number 0) . (pair 15))
(show-pair 15) ==> 15: ((number 3) . (nil ()))

(evl 'y) ==> (0 2)
```

4.2. Rekursion. Als Beispiel für Rekursion die Fakultätsfunktion:

```
(evl '(define f
        (lambda (n)
          (if (= 0 n)
              1
              (* n (f (- n 1)))))))

(evl '(f 5))  ==> 120
(evl '(f 20)) ==> 2432902008176640000
```

Als weiteres Beispiel für Rekursion die Ackermann-Funktion

```
(evl '(define ackermann
        (lambda (x y)
          (if (< y 1) 0
              (if (< x 1) (* 2 y)
```

```
                  (if (< y 2) 2
                      (ackermann (- x 1)
                                 (ackermann x (- y 1))))))))

(evl '(ackermann 1 10)) ==> 1024
(evl '(ackermann 2 4))  ==> 65536
(evl '(ackermann 3 3))  ==> 65536
```

4.3. Höherstufige Prozeduren. Ein Beispiel für höherstufige Prozeduren: Die Prozedur `power` ordnet einer einstelligen Prozedur `f` und einer natürlichen Zahl `n` die Prozedur zu, die durch `n`-malige Iteration von `f` entsteht.

```
(evl '(define power (lambda (f n)
                      (if (< n 1)
                          (lambda (x) x)
                          (lambda (x)
                            (f ((power f (- n 1)) x)))))))

(evl '((power (lambda (x) (+ x x)) 5) 2)) ==> 64

(evl '((power (lambda (x) (* x x)) 4) 2)) ==> 65536

(evl '(((power (lambda (f)
                 (power f 2)) 4) (lambda (x) (+ x x))) 2))
==> 131072
```

Verwandt sind die Iteratoren:

```
(evl '(define it2 (lambda (f) (lambda (x) (f (f x))))))
(evl '(define it3 (lambda (f) (lambda (x) (f (f (f x)))))))
(evl '(define it8
        (lambda (f)
          (lambda (x) (f (f (f (f (f (f (f (f x))))))))))))
(evl '(define s (lambda (x) (+ x 1))))

(evl '(((it3 it2) s) 0)) ==> 8

(evl '(((it8 it2) s) 0)) ==> 256

(evl '((((it3 it2) it2) s) 0)) ==> 256
```

Weitere Beispiele:

```
(evl '(define square (lambda (x) (* x x))))

(evl '(define map1
```

```
        (lambda (f l)
          (if (null? l)
              (quote ())
              (cons (f (car l)) (map1 f (cdr l)))))))

(evl '(map1 square (quote (1 2 3 4 5))))
==> (1 4 9 16 25)

(evl '(map1 (lambda (x) (cons (quote a) x))
            (quote (a b c d))))
==> ((a . a) (a . b) (a . c) (a . d))

(evl '(define p (cons 1 2)))
(evl '(define q p))
(evl '(set-cdr! p 3))
(evl 'q) ==> (1 . 3)
```

4.4. Umgebungen und Zuweisungen. Wir wollen jetzt im Rahmen unseres Interpreters das Beispiel aus Kapitel 3, Abschnitt 2 durchführen.

```
(set! trace? #t)

(evl '(define make-withdraw
        (lambda (balance)
          (lambda (account)
            (if (>= balance account)
                (begin (set! balance (- balance account))
                       balance)
                (quote insufficient-funds))))))

(show-frame 0) ==>
frame number 0 has parent-frame number 0 and bindings
make-withdraw -> (proc (pair 25) 0)
...

(evl '(define f1 (make-withdraw 100)))

largest-used-var-env-index ==> 1

(show-frame 1) ==>
frame number 1 has parent-frame number 0 and bindings
balance -> (number 100)

(evl '(define f2 (make-withdraw 100)))

largest-used-var-env-index ==> 2

(show-frame 2) ==>
```

```
frame number 2 has parent-frame number 0 and bindings
balance -> (number 100)

(show-frame 0) ==>
frame number 0 has parent-frame number 0 and bindings
f2 -> (proc (pair 22) 2)
f1 -> (proc (pair 22) 1)
make-withdraw -> (proc (pair 25) 0)
...

(evl '(f1 50)) ==> 50

largest-used-var-env-index ==> 3

(show-frame 3) ==>
frame number 3 has parent-frame number 1 and bindings
account -> (number 50)

(show-frame 1) ==>
frame number 1 has parent-frame number 0 and bindings
balance -> (number 50)

(evl '(f2 70)) ==> 30

largest-used-var-env-index ==> 4

(show-frame 4) ==>
frame number 4 has parent-frame number 2 and bindings
account -> (number 70)

(show-frame 2) ==>
frame number 2 has parent-frame number 0 and bindings
balance -> (number 30)

(evl '(f2 40)) ==> insufficient-funds

(show-frame 2) ==>
frame number 2 has parent-frame number 0 and bindings
balance -> (number 30)

(evl '(f1 40)) ==> 10

largest-used-var-env-index ==> 6

(show-frame 6) ==>
frame number 6 has parent-frame number 1 and bindings
account -> (number 40)
```

```
(show-frame 1) ==>
frame number 1 has parent-frame number 0 and bindings
balance -> (number 10)
```

Ein etwas veränderter Ablauf liefert

```
(evl '(define make-withdraw
        (lambda (balance)
          (lambda (account)
            (if (>= balance account)
                (begin (set! balance (- balance account))
                       balance)
                (quote insufficient-funds))))))

(evl '(define f1 (make-withdraw 100)))
(evl '(define f2 (make-withdraw 100)))

(evl '(f1 30)) ==> 70
(evl '(f1 13)) ==> 57
(evl '(f2 20)) ==> 80

(evl '(define f3 f1))

(evl '(f3 10)) ==> 47
(evl '(f1 0))  ==> 47
(evl '(f3 50)) ==> insufficient-funds
(evl '(quote (f2 5))) ==> (f2 5)
(evl '(ev (quote (f2 5)))) ==> 75
```

4.5. Zyklische Listen. Ein Beispiel für zyklische Listen:

```
(evl '(define x (quote (a))))
(evl '(define y (cons (quote b) x)))

(evl 'x) ==> (a)
(evl 'y) ==> (b a)
(eval (user-read '(set-cdr! x y)) 0) ==> (pair 0)

(set! trace? #t)
(show-pair 0)  ==>  0: ((symbol a) . (pair 14))
(show-pair 14) ==> 14: ((symbol b) . (pair 0))
```

Man beachte, daß der Versuch der Auswertung von `(evl '(set-cdr! x y))` zu einem Speicherüberlauf führt. Der Grund liegt in dem in `evl` eingebauten Aufruf von `user-display`. Ebenso führt `(evl 'x)` zu einem Speicherüberlauf. Man kann jedoch die zyklische Liste auch auf folgende Weise identifizieren.

```
(evl '(define switch
        (lambda ()
          (begin (set! x (cdr x))
                 (car x)))))

(evl '(switch)) ==> b
(evl '(switch)) ==> a
(evl '(switch)) ==> b
(evl '(switch)) ==> a
```

Literaturverzeichnis

1. Harold Abelson, Gerald Jay Sussman, and Julie Sussman, *Structure and interpretation of computer programs*, 2nd ed., MIT Press, Cambridge, Massachusetts, 1996.
2. Otto Forster, *Analysis 1*, Vieweg, 1999, 5-te Auflage.
3. Richard Kelsey, William Clinger, and Jonathan Rees (Editors), *Revised^5 Report on the Algorithmic Language Scheme*, 1998, http://www.swiss.ai.mit.edu/projects/scheme/.
4. John McCarthy, *Recursive functions of symbolic expressions and their computation by machine*, Communications of the ACM **3** (1960), no. 4, 184–195.

Index