4 Metalinguistische Abstraktion

4.1 Einführung

In diesem Kapitel wollen wir unserem Repertoire der bisherigen Abstraktionstechniken eine weitere mächtige Abstraktion hinzufügen. Hierbei wollen wir Programmiersprachen selbst abstrahieren, d.h. wir wollen eine Technik entwickeln, um dem Programmierer Sprachkonstrukte anzubieten, deren Implementierung wir aber verbergen (was ja der Sinn von Abstraktionen ist).

Die Motivation für ein solches Vorgehen sind vielfältig. Betrachten wir dazu noch einmal generell das Entwickeln von Programmen:

konkreter Problembereich (z.B. Banken, Kunden)
⇓ (Vernachlässigen unnötiger Details)
Modell des Problembereichs (z.B. Bankkonten)
⇓ (Umsetzung mit konkreter Programmiersprache)
Implementierung des Modells

Die Umsetzung in ein konkretes Programm sollte dabei angemessen und nachvollziehbar (*anwendungsorientiert*) sein. Insbesondere sollten die Objekte des Problembereichs auch als Objekte oder Strukturen in der Implementierung dargestellt werden.

Um komplexe Objekte des Problembereichs adäquat zu modellieren, benötigt man gute Abstraktionsmechanismen in der Programmiersprache (vgl. z.B. unseren Schaltkreissimulator). Falls die Abstraktionsmechanismen der Programmiersprache für ein Problem nicht ausreichend sind, kann dies zu einer nicht adäquaten und damit schlechten Modellierung führen. Um dies zu vermeiden, kann man entweder eine neue Sprache realisieren oder eine Sprache um neue, problemadäquate Konstrukte erweitern.

Entwurf und Realisierung neuer Sprachen:

- wichtiges Hilfsmittel bei der Realisierung komplexer Anwendungen
- im allgemeinen aufwändig
- man kann den Aufwand mittels *metalinguistischer Abstraktion* in Grenzen halten
- Idee: realisiere eine neue Sprache durch einen *Interpreter* oder *Evaluator* für diese Sprache
- Evaluator für Sprache PS:
  Eingabe: Programm aus der Sprache PS
  Ausgabe: Berechnungsergebnis bzgl. der Semantik von PS

- Im Folgenden: Ein Evaluator für Scheme geschrieben in Scheme (auch *metazirkulärer Evaluator* genannt)

Warum ist ein metazirkulärer Evaluator für Scheme überhaupt interessant?

- präzise Beschreibung des Auswertungsmechanismus
  (beachte: das Umgebungsmodell war sehr informell beschrieben)
- Erweiterung des Evaluators um
  - neue Sonderformen
  - neue Laufzeitfunktionalität (z.B. Tracing)
  - neue Auswertungsstrategien (z.B. Normalordnung)
  - neue Sprachkonstrukte (z.B. Objekte, Module)

Nachfolgend werden wir einen metazirkulären Evaluator für Scheme entwickeln, der eine präzise Implementierung des Umgebungsmodells darstellt. Aus Gründen der Einfachheit betrachten wir eine Teilmenge von Scheme, allerdings ist der Evaluator leicht auf den vollen Sprachumfang von DrScheme erweiterbar. Außerdem kann man diesen mit geringem Aufwand um die oben erwähnten Aspekte erweitern.

4.2 Quotierung: Programme als Daten

Wir haben schon in Kapitel 2.2.2 gesehen, dass man mittels Quotierung (z.B. `'first`) die Auswertung von Symbolen unterdrücken kann, so dass man mit den Symbolen selbst (und nicht mit deren Bedeutung) rechnen kann. Das Quotieren ist nicht nur auf Symbole, sondern auf beliebige Scheme-Ausdrücke anwendbar. Dies können wir ausnutzen, um Scheme-Programme selbst als Daten darzustellen, die wir z.B. in einem Evaluator verarbeiten können.

> Wenn eine Kombination quotiert wird, dann ist dies äquivalent zur Liste der quotierten Elemente der Anwendung.

Zum Beispiel ist der Scheme-Ausdruck

```
'(a b 99)
```

äquivalent zum Ausdruck

```
(list 'a 'b 99)
```

(quotierte Zahlen sind identisch zu ihren Werten). Ebenso ist der quotierte Ausdruck

```
'(define (quadrat x) (* x x))
```

äquivalent zu

```
(list 'define (list 'quadrat 'x) (list '* 'x 'x))
```

Konkret gibt uns dies die Möglichkeit, Scheme-Programme als normale Listen in Scheme zu verarbeiten, so dass wir keine neuen Datentypen zur Darstellung von Scheme-Programmen in unserem Evaluator benötigen.

Um später die Eingabeschleife eines Scheme-Systems zu realisieren, benutzen wir die Prozedur `read`, die einen Ausdruck von der Standardeingabe liest und diesen so zurück gibt, als wäre dieser quotiert im Scheme-Programm geschrieben worden, wie man an folgendem Dialog sieht (die Zeilen 2 und 5 sind die Eingaben des Benutzers):

```
1 > (read)
2 (a b 99)
3 (list 'a 'b 99)
4 > (read)
5 (define (quadrat x) (* x x))
6 (list 'define (list 'quadrat 'x) (list '* 'x 'x))
```

Da ein quotierter Ausdruck ein ganz normaler Scheme-Ausdruck ist, kann man natürlich einen quotierten Ausdruck wieder quotieren. Dies können wir sehen, wenn wir einen quotierten Ausdruck als Benutzereingabe einlesen:

```
> (read)
'(a b 99)
(list 'quote (list 'a 'b 99))
```

Wie man sieht, ist ein quotierter Ausdruck $'t$ nichts anderes als `(quote` t`)`. Hier ist noch ein weiteres Beispiel:

```
> (equal? 'first (quote first))
true
```

Damit haben wir also die Möglichkeit, auch quotierte Ausdrücke als normale Listen zu verarbeiten. Um zu überprüfen, ob man dies verstanden hat, sollte man sich klar machen, warum die folgenden Ausgaben erfolgen:

```
> (first ''first)
'quote
> (second ''first)
'first
```

Hierbei sollte erwähnt werden, dass die Funktionen `second`, `third`, `fourth` usw. in DrScheme vordefiniert sind durch

```
(define (second l) (first (rest l)))
(define (third  l) (first (rest (rest x))))
(define (fourth l) (first (rest (rest (rest x)))))
...
```

Diese ganz praktischen Funktionen werden wir auch benutzen.

4.3 Ein metazirkulärer Evaluator für Scheme

Syntax von MiniScheme und abstrakte Syntax

Bevor wir an die Implementierung des Evaluators gehen, müssen wir erst einmal festlegen, was der Evaluator verarbeiten kann, d.h. wir müssen die Eingabesprache festlegen, die wir im folgenden als „MiniScheme“ bezeichnen. Wie oben schon erwähnt, werden wir aus Gründen der Einfachheit zunächst einmal eine Teilmenge von Scheme realisieren. Zur Erweiterung auf den vollen Umfang kann dieser Evaluator später nach den hier vorgestellten Prinzipien ergänzt werden.

Wir lassen in MiniScheme folgende Formen von Ausdrücken zu:

Zahlen: Dies können die üblichen Zahlkonstanten sein.

Quotierungen `(quote` a`)`, wobei a ein Ausdruck ist (dies entspricht der Eingabe `'`a).

Variablen: Dies können, wie in Scheme üblich, alle Symbole sein.

Konditionale `(if` b a_1 a_2`)`, wobei b ein Ausdruck ist, der `true` oder `false` liefert.

Abstraktionen `(lambda (`x_1 ... x_n`)` a`)`, wobei die Parameter $x_1, \ldots, x_n$ Variablen und a ein Ausdruck ist.

Definitionen `(define` x a`)`, wobei x eine Variable und a ein Ausdruck ist, oder `(define (`p x_1 ... x_n`)` a`)`, wobei der Prozedurname p ein Symbol, die Parameter $x_1, \ldots, x_n$ Variablen und a ein Ausdruck ist.

Zuweisungen `(set!` x a`)`, wobei x eine schon definierte Variable und a ein Ausdruck ist.

Ausdrucksfolgen `(begin` a_1 ... a_n`)`, wobei a_i Ausdrücke sind.

Lokale Umgebungen `(local (`d_1 ... d_n`)` a`)`, wobei d_i Definitionen und a ein Ausdruck ist.

Applikation `(`p a_1 ... a_n`)`, wobei p und a_i Ausdrücke sind.

Damit enthält MiniScheme alle wesentlichen Elemente von Scheme, bis auf die spezielleren Sonderformen `cond` und `define-struct`.

Um die Mächtigkeit von MiniScheme zu demonstrieren, zeigen wir die Definition einer Konstanten `example-prog`, die eine zulässige Liste von MiniScheme-Definitionen ist und einige bekannte Schemata umfasst:

```
(define example-prog
  '((define (sq x) (* x x))
    (define (fac n) (if (= n 0) 1 (* n (fac (- n 1)))))
```

```
(define (konstr-zaehler)
  (local ((define x 0))
     (lambda () (begin (set! x (+ x 1)) x))))
(define z (konstr-zaehler))))
```

Wie wir gesehen haben, ist der Wert von `example-prog` eine ganz normale (geschachtelte) Scheme-Liste. Damit könnten wir in unserem Evaluator Scheme-Ausdrücke direkt durch die üblichen Listenoperationen verarbeiten. Es ist jedoch besser, auch hier Abstraktionen zu bilden. Statt auf der konkreten Syntax sollte man Operationen zur Verarbeitung einer *abstrakten Syntax* verwenden, um unabhängig von der konkreten Darstellung zu sein.

Beispiel: Wenn wir Zuweisungen (z.B. `(set! x 3)`) auswerten, werden wir, statt direkt auf der Listendarstellung `(list 'set! 'x 3)` zu arbeiten, ein Prädikat `zuweisung?` verwenden, um zu prüfen, ob ein Ausdruck eine Zuweisung ist, und zwei Selektoren `zuweisungs-variable` und `zuweisungs-wert` verwenden, die uns die Variable und den Ausdruck der Zuweisung liefern.

Die Benutzung einer abstrakten Syntax hat den Vorteil, dass wir die konkrete Syntax leicht ändern können (z.B. `(x := 3)` statt `(set! x 3)`), ohne etwas im Kern des Evaluators zu ändern.

Zur Definition der abstrakten Syntax ist das folgende Prädikat nützlich, das wahr ist, falls das zweite Argument eine Liste ist, deren erstes Element das erste Argument ist:

```
(define (liste-mit-symbol? symbol ausdr)
  (if (cons? ausdr)
      (equal? (first ausdr) symbol)
      false))
```

Nun können wir die abstrakte Syntax von Zahlen und speziellen Konstanten, Quotierungen, Variablen, bedingte Ausdrücke, Abstraktionen und Zuweisungen definieren:

```
;;; Zahlen

(define (zahl? ausdr) (number? ausdr))


;;; Quotierungen

(define (quotiert? ausdr) (liste-mit-symbol? 'quote ausdr))

(define (text-der-quotierung ausdr) (second ausdr))


;;; Variablen
```

```
(define (variable? ausdr) (symbol? ausdr))


;;; Zuweisungen

(define (zuweisung? ausdr) (liste-mit-symbol? 'set! ausdr))

(define (zuweisungs-variable ausdr) (second ausdr))

(define (zuweisungs-wert ausdr) (third ausdr))


;;; Bedingte Ausdrücke: (if bedingung ausdruck1 ausdruck2)

(define (bedingt? ausdr) (liste-mit-symbol? 'if ausdr))

(define (if-bedingung ausdr) (second ausdr))

(define (if-konsequente ausdr) (third ausdr))

(define (if-alternative ausdr) (fourth ausdr))

(define (wahr? x) (equal? x true))


;;; Abstraktionen

(define (lambda? ausdr) (liste-mit-symbol? 'lambda ausdr))
```

Bei Definitionen beachten wir, dass es zwei Arten gibt (Variablendefinitionen und Prozedurdefinitionen). Wir stellen aber beide Arten einheitlich dar, indem wir eine Prozedurdefinitionen in eine äquivalenten Lambda-Definition umwandeln:

```
;;; Definitionen

(define (definition? ausdr) (liste-mit-symbol? 'define ausdr))

(define (definitions-variable ausdr)
  (if (variable? (second ausdr))
      (second ausdr)
      ; sonst ist (second ausdr) von der Form (prozedur ...):
      (first (second ausdr))))

(define (definitions-wert ausdr)
```

```
  (if (variable? (second ausdr))
      (third ausdr)
      (list 'lambda
            (rest (second ausdr)) ; formale Parameter
            (third ausdr))))      ; Rumpf
```

Die abstrakte Syntax von Ausdrucksfolgen, lokalen Definitionen und Prozeduranwendungen (Kombinationen) ist ebenfalls einfach definierbar:

```
;;; Folgen von Ausdruecken: (begin ...)

(define (folge? ausdr) (liste-mit-symbol? 'begin ausdr))

(define (folge-ausdruecke ausdr) (rest ausdr))


;;; Lokale Definitionen: (local defs a)

(define (lokal? ausdr) (liste-mit-symbol? 'local ausdr))

(define (lokale-definitionen ausdr) (second ausdr))

(define (lokaler-ausdruck ausdr) (third ausdr))


;;; Prozeduranwendungen

(define (anwendung? ausdr) (cons? ausdr))

(define (operator anw) (first anw))

(define (operanden anw) (rest anw))
```

Im Umgebungsmodell wird ein Prozedurobjekt dargestellt durch ein Paar bestehend aus der Prozedurdefinition und der Umgebung, in der diese Prozedur definiert ist. Daher definieren wir auch eine entsprechende abstrakte Syntax zur Darstellung von Prozeduren:

```
;;; Darstellung von Prozedurobjekten

(define (konstr-prozedur lambda-ausdr umg)
  (list 'prozedur lambda-ausdr umg))

(define (zusammengesetzte-prozedur? proz)
  (liste-mit-symbol? 'prozedur proz))
```

```
(define (parameter proz) (second (second proz)))

(define (prozedur-rumpf proz) (third (second proz)))

(define (prozedur-umgebung proz) (third proz))
```

Darstellung von Umgebungen

Wie wir in Kapitel 3.2 gesehen haben, basiert das Umgebungsmodell auf der Speicherung von Variablenbindungen in Form von Umgebungen. Eine Umgebung ist dabei eine Folge von Rahmen, wobei jeder Rahmen wiederum eine Tabelle von Bindungen, d.h. Zuordnung von Variablen zu Werten ist.

Unser Evaluator implementiert das Umgebungsmodell und muss daher auch Umgebungen und Operationen darauf implementieren. Die wichtigsten Operationen auf Umgebungen sind:

- `(nachsehen-variablenwert var umg)`: liefert den Wert einer Variablen bezüglich einer Umgebung.
- `(erweitern-umgebung variablen werte basis-umg)`: Liefert eine neue Umgebung, die eine gegebene Basisumgebung um einen neuen Rahmen erweitert, in dem die Zuordnung der gegebenen Variablen zu Werten eingetragen ist. Zum Beispiel wird die Erweiterung einer Umgebung `umg` um ein neuen Rahmen mit der Zuordnung von `x` zu `1` und `y` zu `13` durch die Kombination

  ```
  (erweitern-umgebung (list 'x 'y) (list 1 13) umg)
  ```

  berechnet.
- `(define-variable! var wert umg)`: Fügt die Bindung der Variablen zum gegebenen Wert in den ersten Rahmen der Umgebung hinzu.
- `(set-variablenwert! var wert umg)`: Setzt die existierende Bindung der Variablen `var` neu.

Um diese Operationen zu implementieren, müssen wir eine konkrete Darstellung von Umgebungen festlegen. Hierzu führen eine Struktur `umgebung` zur Darstellung von Umgebungen als Folge von Rahmen ein:

```
;;; Darstellung von Umgebungen

(define-struct umgebung (erster-rahmen rest-rahmen))

(define (erster-rahmen umg) (umgebung-erster-rahmen umg))
```

```
(define (rest-rahmen umg) (umgebung-rest-rahmen umg))

(define (keine-rahmen-mehr? umg) (empty? umg))

(define (hinzufuegen-rahmen rahmen umg) (make-umgebung rahmen umg))

(define (set-erster-rahmen! umg neuer-rahmen)
  (set-umgebung-erster-rahmen! umg neuer-rahmen))
```

Für die Darstellung der Bindungsrahmen selbst benutzen wir, wie auch schon bei der früheren Implementierung von Tabellen, eine Assoziationsliste von Bindungen:

```
;;; Darstellung von Bindungsrahmen

(define (konstr-rahmen variablen werte)
  (cond ((and (empty? variablen) (empty? werte)) empty)
        ((empty? variablen)
         (error 'konstr-rahmen "Zu viele Werte angegeben"))
        ((empty? werte)
         (error 'konstr-rahmen "Zu wenige Werte angegeben"))
        (else
         (cons (konstr-bindung (first variablen) (first werte))
               (konstr-rahmen (rest variablen) (rest werte))))))

(define (hinzufuegen-bindung bindung rahmen)
  (cons bindung rahmen))

(define (bindung-in-rahmen var rahmen)
  (suche-bindung var rahmen))

; Suche in Assoziationsliste:
(define (suche-bindung var bindungen)
  (cond ((empty? bindungen) keine-bindung)
        ((equal? var (bindungsvariable (first bindungen)))
         (first bindungen))
        (else (suche-bindung var (rest bindungen)))))

(define (bindung-gefunden? b)
  (not (equal? b keine-bindung)))

(define keine-bindung empty)
```

```
;;; Darstellung von Bindungen

(define-struct bindung (variable wert))

(define (konstr-bindung variable wert)
  (make-bindung variable wert))

(define (bindungsvariable bindung)
  (bindung-variable bindung))

(define (bindungswert bindung)
  (bindung-wert bindung))

(define (set-bindungswert! bindung wert)
  (set-bindung-wert! bindung wert))
```

Mit dieser Implementierung sind wir in der Lage, die oben angegebenen Operationen auf Umgebungen zu implementieren:

```
;;; Operationen auf Umgebungen

(define (nachsehen-variablenwert var umg)
  (local ((define b (bindung-in-umg var umg)))
    (if (bindung-gefunden? b)
        (bindungswert b)
        (error var "Ungebundene Variable"))))

(define (bindung-in-umg var umg)
  (if (keine-rahmen-mehr? umg)
      keine-bindung
      (local ((define b (bindung-in-rahmen var (erster-rahmen umg))))
        (if (bindung-gefunden? b)
            b
            (bindung-in-umg var (rest-rahmen umg))))))

(define (erweitern-umgebung variablen werte basis-umg)
  (hinzufuegen-rahmen (konstr-rahmen variablen werte) basis-umg))

(define (define-variable! var wert umg)
  (local ((define b (bindung-in-rahmen var (erster-rahmen umg))))
    (if (bindung-gefunden? b)
        (error var "Redefinition nicht erlaubt!")
        (set-erster-rahmen!
          umg
```

```
          (hinzufuegen-bindung (konstr-bindung var wert)
                               (erster-rahmen umg))))))

(define (set-variablenwert! var wert umg)
  (local ((define b (bindung-in-umg var umg)))
    (if (bindung-gefunden? b)
        (set-bindungswert! b wert)
        (error var "Ungebundene Variable"))))
```

Auswertungsregeln des Umgebungsmodells

Mit der abstrakten Syntax und den Umgebungsoperationen haben wir alle Elemente zur Verfügung, um die Auswertungsregeln des Umgebungsmodells direkt umzusetzen. Man beachte, dass wir mit dieser Umsetzung nun auch eine präzise Definition des Umgebungsmodells angeben.

Die Kern des Evaluators ist eine Prozedur `auswerten`, die einen gegebenen Ausdruck in einer gegebenen Umgebung auswertet. Diese Prozedur ist definiert durch eine Fallunterscheidung über die Struktur des Ausdrucks:

```
(define (auswerten ausdr umg)
  (cond
    ((zahl?       ausdr) ausdr)
    ((quotiert?   ausdr) (text-der-quotierung ausdr))
    ((variable?   ausdr) (nachsehen-variablenwert ausdr umg))
    ((definition? ausdr) (auswerten-definition ausdr umg))
    ((zuweisung?  ausdr) (auswerten-zuweisung ausdr umg))
    ((lambda?     ausdr) (konstr-prozedur ausdr umg))
    ((bedingt?    ausdr) (auswerten-bedingt ausdr umg))
    ((folge?      ausdr) (auswerten-folge
                                  (folge-ausdruecke ausdr) umg))
    ((lokal?      ausdr) (auswerten-lokale-definitionen ausdr umg))
    ((anwendung?  ausdr) (anwenden
                            (auswerten (operator ausdr) umg)
                            (liste-der-werte (operanden ausdr) umg)))
    (else (error 'auswerten "Unbekannter Ausdruckstyp"))))
```

Zur übersichtlicheren Struktur des Evaluators definieren wir die Auswertung der unterschiedlichen Ausdrucksformen durch verschiedene Prozeduren:

```
(define (auswerten-definition ausdr umg)
  (define-variable!
      (definitions-variable ausdr)
      (auswerten (definitions-wert ausdr) umg)
```

```
      umg))

(define (auswerten-zuweisung ausdr umg)
  (set-variablenwert! (zuweisungs-variable ausdr)
                      (auswerten (zuweisungs-wert ausdr) umg)
                      umg))

(define (auswerten-bedingt ausdr umg)
  (if (wahr? (auswerten (if-bedingung ausdr) umg))
      (auswerten (if-konsequente ausdr) umg)
      (auswerten (if-alternative ausdr) umg)))

(define (auswerten-folge ausdruecke umg)
  (if (empty? (rest ausdruecke)) ; letzter Ausdruck?
      (auswerten (first ausdruecke) umg)
      (begin (auswerten (first ausdruecke) umg)
             (auswerten-folge (rest ausdruecke) umg))))

(define (auswerten-lokale-definitionen ausdr umg)
  (local ((define neue-umg (erweitern-umgebung empty empty umg)))
    (begin (auswerten-folge (lokale-definitionen ausdr) neue-umg)
           (auswerten (lokaler-ausdruck ausdr) neue-umg))))

(define (anwenden prozedur argumente)
  (cond ((elementare-prozedur? prozedur)
         (anwenden-elementare-prozedur prozedur argumente))
        ((zusammengesetzte-prozedur? prozedur)
         (auswerten (prozedur-rumpf prozedur)
                    (erweitern-umgebung
                       (parameter prozedur)
                       argumente
                       (prozedur-umgebung prozedur))))
        (else
         (error 'anwenden "Unbekannter Prozedurtyp"))))

(define (liste-der-werte ausdruecke umg)
  (if (empty? ausdruecke)
      empty
      (cons (auswerten (first ausdruecke) umg)
            (liste-der-werte (rest ausdruecke)
                             umg))))
```

Damit ist der Kern des Evaluators, der das Umgebungsmodell implementiert, fertig.

Die Eingabeschleife

Um den Evaluator auch praktisch anzuwenden und damit ein vollständiges interaktives System für MiniScheme zu realisieren, fehlt noch eine Eingabeschleife zur Eingabe und Auswertung von Ausdrücken. Außerdem fehlt noch die Implementierung elementarer Prozeduren, die in jedem Scheme-System vordefiniert sind. Um letzteres zu realisieren, müssen wir elementare Prozeduren von benutzerdefinierten Prozeduren unterscheiden. Zu diesem Zweck stellen wir elementare Prozedurobjekte in der Form `'(elementar` n`)` dar, wobei n der Name der elementaren Prozedur ist. Somit definieren wir die Liste der Namen vordefinierten elementarer Prozeduren (`namen-elementarer-prozeduren`), die dazugehörigen elementare Prozedurobjekte, die wir aus der Namensliste durch Anwendung der vordefinierte Prozedur `map` (dies ist identisch zu `map-list` aus Kapitel 2.3.3) berechnen, die abstrakte Syntax elementarer Prozeduren und die Anwendung einer elementaren Prozedur auf eine Liste von Werten wie folgt:

```
(define namen-elementarer-prozeduren
  '(first rest cons empty? + - * / =
        ;;** Füge die Namen weiterer elementarer Prozeduren hinzu
        ))

(define objekte-elementarer-prozeduren
  (local ((define (elementare-proz name)
            (list 'elementar name)))
     (map elementare-proz namen-elementarer-prozeduren)))

(define (elementare-prozedur? proz)
  (liste-mit-symbol? 'elementar proz))

(define (elementare-id proz) (second proz))

(define (anwenden-elementare-prozedur proz argliste)
  (local ((define p (elementare-id proz))
          (define arg1 (first argliste)))
    (cond ((equal? p 'first) (first arg1))
          ((equal? p 'rest) (rest arg1))
          ((equal? p 'cons) (cons arg1 (second argliste)))
          ((equal? p 'empty?) (empty? arg1))
          ((equal? p '+) (+ arg1 (second argliste)))
          ((equal? p '-) (- arg1 (second argliste)))
          ((equal? p '*) (* arg1 (second argliste)))
          ((equal? p '/) (/ arg1 (second argliste)))
          ((equal? p '=) (= arg1 (second argliste)))
          ;;** weitere elementare Prozeduren
          (else (error 'anwenden-elementare-prozedur
```

```
                          "Unbekannte elementare Prozedur")))))
```

Der Ausdruck, der durch den Benutzer des Systems eingegeben wird, wird immer bezüglich der globalen Umgebung ausgewertet, in der zu Beginn die vordefinierten Prozeduren und Namen eingetragen sind. Zu diesem Zweck definieren wir die globale Umgebung durch Initialisierung einer leeren Umgebung mit den Namen elementarer Prozeduren und den üblichen Basiskonstanten:

```
(define (einrichten-umgebung)
  (local ((define initial-umg
            (erweitern-umgebung namen-elementarer-prozeduren
                                objekte-elementarer-prozeduren
                                empty)))
    (begin (define-variable! 'true true initial-umg)
           (define-variable! 'false false initial-umg)
           (define-variable! 'empty empty initial-umg)
           initial-umg)))

(define die-globale-umgebung (einrichten-umgebung))
```

Damit müssen wir nur noch die Eingabeschleife definieren, die jeweils einen Ausdruck einliest, in der globalen Umgebung auswertet und das Ergebnis ausdruckt:

```
(define (treiber-schleife)
  (begin
    (newline)
    (display "MiniScheme> ")
    (benutzer-print (auswerten (read) die-globale-umgebung))
    (treiber-schleife)))

(define (benutzer-print objekt)
  (if (zusammengesetzte-prozedur? objekt)
      (display (list 'zusammengesetzte-prozedur
                     (parameter objekt)
                     (prozedur-rumpf objekt)
                     'prozedur-umgebung))
      (display objekt)))
```

Zum angenehmeren Austesten können wir in die globale Umgebung auch ein initiales Programm eintragen, das in dem Evaluator-Programm als Konstante definiert ist, wie z.B. die oben definierte Konstante `example-prog`. Dazu definieren wir uns eine Prozedur

```
(define (starte-programm prog)
  (begin (auswerten-folge prog die-globale-umgebung)
```

```
        (treiber-schleife)))
```

so dass wir unseren Evaluator mit

```
(runprog example)
```

starten können. Hier ist ein kleiner Beispieldialog mit unserem Evaluator (die Benutzereingaben sind jeweils umrandet):

```
MiniScheme> (sq (fac 4))
576
MiniScheme> (fac (sq 6))
371993326789901217467999448150835200000000
MiniScheme> z
(zusammengesetzte-prozedur () (begin (set! x (+ x 1)) x) prozedur-umgebung)
MiniScheme> (z)
1
MiniScheme> (z)
2
MiniScheme>
```

Wie man sieht, funktioniert unser selbst geschriebener Evaluator wie ein ganz normales Scheme-System. Im Gegensatz zu einem vorgegebenen Scheme-System haben wir nun aber die Möglichkeit, mit relativ wenig Aufwand den Umfang der Eingabesprache unseres Evaluators zu erweitern. Dies wollen wir nachfolgend erläutern.

Übungsaufgaben

1. Erweitere den Evaluator um weitere vordefinierte elementare Prozeduren, wie z.B. `<`, `>`, `<=`, `>=`, `not`, `and`, `or`, `list` (man beachte, dass die letzten drei Prozeduren beliebig viele Argumente haben können).
2. Erweitere MiniScheme und den Evaluator um die Sonderformen `(cond ...)` und `(define-struct ...)`.
3. Erlaube als Syntax für die Zuweisung auch die Sonderform `(var := ausdruck)`.

4.4 Module

Die wichtigste Technik zur Beherrschbarkeit großer Systeme ist die Zerlegung in überschaubare Einheiten. Wenn wir viele solcher Einheiten entwickeln, ist es wichtig,

- den *Namensraum* zu strukturieren um
- *Namenskonflikte* zu vermeiden.

Beispiel: Zwei Programmierer implementieren in verschiedenen Programmen auf unterschiedlichen Datentypen (z.B. komplex, rational) die Prozedur `add`. Wenn wir nun diese Teilprogramme zu einem Gesamtprogramm zusammenfügen und in diesem Programm den Namen `add` verwenden, stellt sich die Frage, welche `add`-Implementierung gemeint ist. Dies ist ein *Namenskonflikt*.

Hierzu gibt es folgende Lösungen:

- Wähle eindeutige Namen (z.B. `add-komplex`, `add-rational`):
  - fehleranfällig
  - umständlich
- Blockstruktur: Schachtelung in lokalen Prozeduren. Dies ist nur für kleinere Programmeinheiten geeignet.
- Module:
  - logisch zusammenhängende Einheiten von Prozeduren und Daten
  - Implementierung muss außen nicht bekannt sein (d.h. ein Modul entspricht einem abstrakten Datentyp und bildet damit eine Abstraktionsbarriere)
  - unabhängig implementierbar (nur mittels Schnittstellen anderer Module)
  - beliebige Wahl lokaler Namen ohne Konflikte

Realisierung von Modulen

Im Prinzip haben wir schon in Kapitel 3.1 eine Möglichkeit zur Realisierung von Modulen kennengelernt: die Nachrichtenweitergabe.

Beispiel: Modul zur Kontenbearbeitung:

```
(define (konstr-konto wert)
  (local ((define kontostand wert)

          (define (abheben betrag)
            (if (>= kontostand betrag)
              (begin
                (set! kontostand (- kontostand betrag))
                kontostand)
              "Deckung unzureichend"))

          (define (einzahlen betrag)
            (begin
              (set! kontostand (+ kontostand betrag))
              kontostand))

          (define (zuteilen m)
            (cond ((equal? m 'abheben) abheben)
```

```
                ((equal? m 'einzahlen) einzahlen)
                (else
                 (error 'konstr-konto "Unbekannte Forderung")
                 ))))
  zuteilen))
```

Dieses Modul besteht aus

- lokalen Daten (`kontostand`), und
- lokalen Prozeduren (`abheben`, `einzahlen`)

Hierbei gibt es keine Namenskonflikte der lokalen Namen mit anderen Namen wegen der lokalen Definition (Blockstruktur). Die Schnittstelle dieses Moduls bilden die Nachrichten `'abheben` und `'einzahlen`, weil man damit auf die Funktionalität eines Kontos zugreifen kann.

Eine solche Modellierung ist sinnvoll, weil das Konto einen lokalen Zustand hat. Für Module ohne lokalen Zustand (d.h. eine Sammlung von Prozeduren) ist die Realisierung mittels Nachrichtenweitergabe möglich, aber sehr umständlich, wenn man nur die Namen kontrollieren will. Besser wäre es, die Umgebungen, d.h. die *Namensräume* selbst zu kontrollieren.

Beispiel: Die Prozeduren für die komplexe Arithmetik werden in einer Umgebung mit dem Namen `komplex` zusammengefasst, und die Prozeduren für die rationale Arithmetik werden in einer anderen Umgebung mit dem Namen `rational` zusammengefasst. Dann kann die Namensauflösung durch Angabe der Umgebung erfolgen: z.B. wird eine Kombination `(add` t_1 t_2`)` in der Umgebung `komplex` oder in der Umgebung `rational` ausgewertet.

Zur Realisierung dieser Idee führen wir eine neue Sonderform

```
(eval Ausdruck Umgebung)
```

ein, die einen Ausdruck in einer definierten Umgebung auswertet. Zur Konstruktion von Umgebungen führen wir die Sonderform

```
(make-environment
  (define ...)
  ⋮
  (define ...))
```

ein, die als Ergebnis eine neue Umgebung liefert, die die lokalen Definitionen beinhaltet.

Hier ist ein Beispiel für die beabsichtige Funktionsweise dieser Sonderformen:

```
> (define u1 (make-environment (define x 2)))
> (define u2 (make-environment (define x 5)))
> x
```

```
x: Ungebundene Variable
> (eval x u1)
2
> (eval x u2)
5
```

Als weiteres Beispiel betrachten wir die Implementierung eines Moduls mit mathematischen Funktionen:

```
(define math-funktionen
  (make-environment
    (define (fakultaet n) ...)
    (define (potenz b n) ...)
    (define (wurzel x)
      (local ((define (verbessern y) ...)
              ...)))))
```

Wir können dann dieses Modul z.B. anwenden durch

```
> (eval (potenz 2 8) math-funktionen)
256
```

Wir können auch in einem Modul definierte Namen „globalisieren“, wie das folgende Beispiel zeigt:

```
> (define potenz (eval potenz math-funktionen))
> (potenz 2 7)
128
```

Hierbei ist der Wert des Ausdrucks `(eval potenz math-funktionen)` eine Prozedur in der Umgebung `math-funktionen`).

Da diese Sonderformen kein Standard in Scheme sind (es gibt aber Scheme-System mit Modulen), müssen wir diese selbst implementieren, was aber durch die Erweiterung unseres Evaluators nicht schwierig ist. Hier müssen wir im wesentlich die Prozedur `auswerten` etwas erweitern:

1. Füge in der Fallunterscheidung in `auswerten` eine neue Klausel ein:

   ```
   ((eval? ausdr) (auswerten (eval-ausdruck ausdr)
                             (auswerten (eval-umg ausdr) umg)))
   ```

   Hierbei sind `eval?`, `eval-ausdruck` und `eval-umg` entsprechende Prozeduren für die abstrakte Syntax von Ausdrücken der Form `(eval? ...)`.

2. In ähnlicher Weise wie für Prozeduren führen wir die abstrakte Syntax zur Darstellung des Konstrukts `(make-environment ...)` ein und erweitern `auswerten` um eine Klausel, die folgendes implementiert (ähnlich wie `auswerten-lokale-definitionen`):

- Auswertung der Definitionsfolge bzgl. der aktuellen Umgebung
- Rückgabe der erweiterten Umgebung als Ergebnis

Hier wäre es besser, Umgebungen als neuen Datentyp (analog zu `konstr-prozedur`) einzuführen, um fehlerhafte Anwendungen von Umgebungen zu melden.

4.5 Auswertung in Normalordnung

Wir skizzieren nun, wie man unseren Evaluator so erweitern kann, dass er Programme in Normalordnung statt in applikativer Ordnung auswertet.

Die Idee der Normalordnung ist, Prozedurargumente erst auszuwerten, wenn deren Werte benötigt werden, z.B. von elementaren Prozeduren. Hierdurch können unnötige und auch unendliche Auswertungen vermieden werden.

Betrachten wir dazu die Definition

```
(define (f a b)
  (if (= a 0) 1 b))
```

und den Aufruf

```
> (f 0 (/ 1 0))
```

- Applikative Auswertung: Laufzeitfehler ("division by zero") wegen Auswertung von `(/ 1 0)`.
- Normalordnung: Ergebnis `1`

Beachte:

> Jede Sprache hat mindestens eine Funktion, die in Normalordnung ausgewertet wird (Fallunterscheidung), sonst wären rekursive Prozeduren nicht formulierbar!

Beispiel:

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

Falls `if` seine Argumente *vor* der Anwendunge auswertet, würde der Aufruf `(fak 0)` nicht terminieren:

```
(fak 0)  ->  (if (= 0 0) 1 (* (fak (- 0 1)) 0))
         ->  (if true 1 (* (fak (- 0 1)) 0))
         ->  (if true 1 (* (fak -1) 0))
         ->  (if true 1 (* (if (= -1 0)
                               1
                               (* (fak (- -1 1)) -1)) 0))
         ->  ...
```

Im Prinzip könnten wir alle Funktionsaufrufe in Normalordnung abarbeiten (dies wird z.B. in der Programmiersprache Haskell gemacht). Dies hätte den Nachteil eines möglichen Effizienzverlustes und eines unklaren Ablaufs von Seiteneffekten (`set!`). Daher realisieren wir hier eine andere Idee: Wir erlauben die explizite Angabe von verzögerten Argumenten bei einer Prozedurdeklaration (Parameter `(delay ...)`), d.h. wir können die applikative und die normale Auswertung mischen.

Ein Beispiel soll die Anwendung dieser Idee zeigen. Wir könnten mit dieser Erweiterung die obige Funktion `f` mit verzögerter Argumentauswertung wie folgt definieren (hier ist zu beachten, dass `(delay a)` eine Annotation zur Verzögerung des Parameters `a` und kein Prozeduraufruf ist!):

```
(define (f (delay a) (delay b))
  (if (= a 0) 1 b))
```

Damit würde der Aufruf `(+ 2 (f 0 (/ 1 0)))` zu `3` auswerten.

Um dies in unserem Evaluator zu realisieren, müssten nur folgende Modifikationen gemacht werden:

- Bei `auswerten`: Betrachte vor(!) Auswertung einer Anwendung die Prozedurdefinition und seine formalen Parameter:
  - Falls das Argument nicht verzögert ist (d.h. der formale Parameter ist ein Symbol), dann wird das Argument wie üblich ausgewertet.
  - Falls das Argument verzögert ist (d.h. der formale Parameter hat die Form `(delay ...)`, erzeuge ein Objekt, mit dem man dieses Argument später auswerten kann: erzeuge einen *Abschluss* (*closure*, *thunk*), d.h. ein Paar bestehend aus einem Ausdruck und einer Umgebung für die Auswertung dieses Ausdrucks (dies entspricht einer Prozedur ohne Parameter).
- Werte Abschlussobjekte aus, wenn sie „gebraucht“ werden: z.B. „brauchen“ elementare Prozeduren ihre Argumente, d.h. ändere `anwenden-elementare-prozedur`: Werte vor der Anwendung jedes Abschlussargument aus.

Die Programmierung der genauen Details ist recht einfach ($\rightsquigarrow$ Übung).