

Kapitel 5

Datenabstraktion

Bei der Programmierung taucht immer wieder die Frage der *Datenrepräsentation* auf: „Wie lassen sich die Größen des Problems als Werte im Programm darstellen?“ Diese Frage ist häufig schon schwer genug zu beantworten. Schlimmer noch, während der Entwicklungszeit eines Programms kann es notwendig werden, die Datenrepräsentation zu ändern. Dies kann schwerwiegende Auswirkungen auf ein Programm haben.

Gut geschriebene Programme sind allerdings gegen solche Änderungen von Repräsentationen weitgehend immun. Ein solches Programm erlaubt die Änderung von Datenrepräsentationen, ohne daß gleich der gesamte Code neu geschrieben werden muß. Die entscheidenden Programmier Techniken für solche Programme sind die Einführung von *Datenabstraktion* und die Verwendung von *Typen*.

5.1 Ein-/Ausgabe und sequentielle Auswertung

Die bisherigen Programme haben recht spärlich mit dem Benutzer interagiert: Sie wurden mit einer Eingabe versorgt, heraus kam eine Ausgabe. Für interaktive Programme ist dies nicht genug. Einige weitere Prozeduren und ein weiteres Sprachmittel erlauben *Ein- und Ausgabe* und damit Interaktion mit dem Benutzer.

Die Prozedur `display` druckt die Repräsentation ihres Parameters in der REPL aus. Der Effekt ist gar nicht so einfach zu demonstrieren, aber immerhin:

```
> (display 5)
5
```

Die Prozedur `newline` (ohne Parameter) gibt einen Zeilenvorschub aus.

```
> (newline)
```

Um wirklich sinnvolle Ein-/Ausgabe zu gestalten, ist es notwendig, mehrere davon hintereinander vorzunehmen. Die neue Spezialform `begin` erlaubt genau dies: Sie wertet jeden ihrer Operanden nacheinander aus und verwirft ihre Werte *bis auf den letzten*. Dieser letzte Wert wird zum Wert der `begin`-Form:

```
<expression> ::= (begin <sequence>)
<sequence> ::= <expression>+
```

Also:

```
> (begin 1 2 3 4 5)
5
> (begin (display 23) (newline) (display 17) (newline))
23
17
```

`begin` umschließt implizit den Rumpf von Abstraktionen und damit auch `let`-Ausdrücken:

```
> (let ((x 1)) (display x) (newline) (display (+ x x)) (newline))
1
2
```

Auch wenn diese Prozeduren zunächst (besonders für Pascal- und C-Programmierer) harmlos aussehen, ist mit `display` und `newline` ein entscheidender Faktor mit in die Programmierung gekommen: Bei allen Ausdrücken bisher hat nur ihr *Wert* interessiert, bei `display` und `newline` interessiert zum ersten Mal der Wert überhaupt nicht, sondern lediglich der *Effekt*. Mit Effekten kommt plötzlich *Reihenfolge* ins Spiel, die vorher zwar festgelegt, aber weitgehend unwichtig war.

Gelegentlich ist es wünschenswert, einen Text auszugeben. Zu diesem Zweck kann der Text in Anführungszeichen eingeschlossen an `display` übergeben werden:

```
> (display "C est si bon.")
C est si bon.
```

Soweit zur Ausgabe. Zur Eingabe kann die Prozedur `read` verwendet werden; sie wartet auf die Eingabe der Repräsentation eines Werts und gibt den entsprechenden Wert zurück:

```
> (read)
5

5
> (read)
#t

#t
> (read)
(1 . 2)

(1 . 2)
```

Eine weitere praktische Prozedur ist `error`. Sie hat beliebig viele Parameter, von denen der erste in Anführungszeichen eingeschlossen sein müssen. Bei der Auswertung von `error` werden die Repräsentationen der Parameter nacheinander ausgegeben; danach wird das Programm beendet. `Error` wird dazu benutzt, um das Programm zu beenden, wenn durch Fehler oder Fehleingaben eine sinnvolle weitere Ausführung nicht mehr möglich ist.

5.2 Syntaktischer Zucker für Conditionals

Für geschachtelte Conditionals gibt es `cond`:

```
(expression) ::= (cond (cond clause)+)
(expression) ::= (cond (cond clause)* (else (sequence)))
(cond clause) ::= ((test) (sequence))
```

Intuitiv wertet `cond` nacheinander alle Tests aus; sobald einer `#t` ergibt, reduziert sich der `cond`-Ausdruck zur entsprechenden rechten Seite. `Else` verhält sich wie ein Test, der nie fehlschlägt.

Formal wird `cond` ebenfalls intern in `if` übersetzt, und zwar nach folgendem induktiven Schema:

```
(cond ((t0 s0) (t1 s2) ...))
⇒
(if t0 s0 (cond ((t1 s2) ...)))

(cond ((else s))
⇒
s
```

Mit diesen Mitteln könnte die naive Version von `fib` so aussehen:

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

5.3 Symbole und Gleichheit

Manchmal signalisieren Werte Dinge, die sich durch Worte besser beschreiben lassen als durch Zahlen oder Booleans. Zu diesem Zweck (und noch anderen) gibt es in Scheme den Datentyp der *Symbole*, die als externe Repräsentation einen Namen haben. Wie die leere Liste auch sehen Literale und externe Repräsentation bei Symbolen unterschiedlich aus; die Literale fangen mit einem Apostroph an, das bei der externen Repräsentation fehlt:

```
> 'hans-otto
hans-otto
```

Dabei kann ein Symbol jeden Namen tragen, den auch eine Variable tragen kann und umgekehrt. Entscheidend ist aber der Unterschied zwischen Symbolen und Variablen:

```
> (define hans-otto 'maria-berta)
> hans-otto
maria-berta
> 'hans-otto
hans-otto
```

Zum Vergleich von Symbolen kann das Prädikat `equal?` verwendet werden:

```
> (equal? 'hans-otto 'maria-berta)
#f
> (equal? 'hans-otto 'hans-otto)
#t
```

Tatsächlich kann `equal?` für das Testen der Gleichheit *beliebiger* Objekte verwendet werden. (Die Erklärung des Verhaltens von `equal?` ist allerdings dennoch überraschend subtil. Mehr dazu später.)

5.4 Ein Programm für einen Getränkeautomaten

Die Vorteile der Datenabstraktion kommen erst bei Programmen einer bestimmten Größe zum Tragen. Darum will erst einmal ein Beispiel geschrieben sein, dessen Bedeutung für die Fragen der Datenrepräsentation und -abstraktion dann später klarwerden wird.

Ziel ist die Steuerung eines Getränkeautomaten. Dieser Automat führt verschiedene Getränkesorten und soll die Preise von Bestellungen richtig verarbeiten. Mangels der echten Hardware soll das Programm in der REPL einen Dialog mit dem Benutzer führen. Dieser Dialog könnte so aussehen, wobei die Benutzereingaben durch *Schrägschrift* ausgezeichnet sind:

```
> (drink-machine prices inventory)
The inventory:
drink: coke stock: 300
drink: fanta stock: 100
drink: karamalz stock: 100
drink: black-bush stock: 100
What drink?
black-bush
How many cans?
200
Sorry, we don't have enough black-bush in stock.
The inventory:
drink: coke stock: 300
drink: fanta stock: 100
drink: karamalz stock: 100
drink: black-bush stock: 100
What drink
coke
How many cans?
500
Sorry, we don't have enough coke in stock.
```

Ist der Bestand irgendwann erschöpft, so soll das Programm ausdrucken:

```
drink: coke stock: 0
drink: fanta stock: 0
drink: karamalz stock: 0
drink: black-bush stock: 0
I need a drink, too.
```

Die Preisliste und das Inventar sollen zunächst als Assoziationslisten repräsentiert werden, für das obige Beispiel also folgendermaßen:

```
(define prices
  (list (cons 'coke 100)
        (cons 'fanta 100)
        (cons 'karamalz 120)
        (cons 'black-bush 250)))

(define inventory
  (list (cons 'coke 300)
        (cons 'fanta 100)
        (cons 'karamalz 200)
        (cons 'black-bush 100)))
```

Ein solches Programm läßt sich — ausgehend von dem Beispielprotokoll — am einfachsten mit Hilfe von Wunschdenken schreiben. Die `drink-machine`-Prozedur muß zunächst das Inventar ausdrucken:

```
(define drink-machine
  (lambda (prices inventory)
    (display-inventory inventory)
```

Die Prozedur `display-inventory` ist — wie viele der anderen Prozeduren, die von `drink-machine` aufgerufen werden, noch nicht geschrieben. `Drink-machine` muß als nächstes überprüfen, ob der Bestand erschöpft ist:

```
(if (empty? inventory)
    (announce-the-end)
```

Für den glücklichen Fall, daß noch Getränke vorhanden sind, kann das Programm eine Bestellung akzeptieren:

```
(let ((order (accept-order)))
```

Die Bestellung muß daraufhin auf Erfüllbarkeit getestet werden. Wenn sie erfüllbar ist, wird sie bestätigt und das Geld wird eingesammelt:

```
(if (order-satisfiable? order inventory)
    (begin
      (confirm-order order)
      (let* ((price (order-price order prices))
            (payment (accept-payment price)))
        (display-change price payment))
```

`Order-price` ist eine Hilfsprozedur, welche den Preis einer Bestellung aus Getränkesorte, Dosenanzahl und Preisliste berechnet. Ihre Definition wird weiter unten erläutert. Nun muß die Bestellung noch vom Bestand abgezogen werden und es geht von vorn los:

```
(drink-machine prices
  (apply-order order inventory)))
```

Im anderen Zweig des `if` von oben wird die Bestellung zurückgewiesen und es geht mit unverändertem Bestand von vorn los:

```
(begin
  (display-rejection order inventory)
  (drink-machine prices inventory))))))
```

Die erste Prozedur, die von `drink-machine` aufgerufen wird, ist `display-inventory`. Sie benutzt die eingebaute Prozedur `for-each`. Diese hat als Argumente eine Prozedur und eine Liste. Diese Prozedur wird auf jedes Element der Liste nacheinander angewendet. Der Rückgabewert von `for-each` ist unspezifiziert. `For-each` funktioniert also ähnlich wie `map`, verwirft aber die Rückgabewerte der Prozedur:

```
(define display-inventory
  (lambda (inventory)
    (display "The inventory: ")
    (newline)
    (for-each (lambda (inventory-entry)
                (display "drink: ")
                (display (car inventory-entry))
                (display " stock: ")
```

```

      (display (cdr inventory-entry))
      (newline))
  inventory)))

```

Die Prozedur `empty?` testet, ob der Lagerbestand leer ist. Es benutzt dazu die `every?`-Prozedur aus dem letzten Kapitel, die prüft, ob ein Prädikat für alle Elemente einer Liste erfüllt ist:

```

(define empty?
  (lambda (inventory)
    (every? (lambda (inventory-entry)
              (= 0 (cdr inventory-entry)))
            inventory)))

```

Die Prozedur `announce-the-end` — ebenfalls von `drink-machine` aufgerufen — gibt das traurige Ergebnis der Inventur zurück:

```

(define announce-the-end
  (lambda ()
    (display "I need a drink, too.")
    (newline)))

```

Als nächstes ist die Prozedur `accept-order` an der Reihe. Sie soll die Fragen `What drink?` und `How many cans?` stellen und jeweils eine Antwort vom Benutzer erfragen. Für diese Aufgabe (Frage stellen, Antwort abwarten) benutzt `accept-order` die Hilfsprozedur `prompt`:

```

(define accept-order
  (lambda ()
    (let* ((product (prompt "What drink?"))
           (number (prompt "How many cans?")))
      (cons product number))))

```

```

(define prompt
  (lambda (prompt-string)
    (display prompt-string)
    (newline)
    (read)))

```

Eine Bestellung ist — per `accept-order` — ein Paar aus Getränkesorte und Anzahl der bestellten Dosen. Die Prozedur `order-satisfiable?` überprüft, ob eine solche Bestellung aus einem gegebenen Bestand erfüllbar ist. Sie benutzt dazu die Hilfsprozedur `inventory-drink-entry`

```

(define order-satisfiable?
  (lambda (order inventory)
    (let* ((requested-drink (car order))
           (requested-number-of-cans (cdr order))
           (number-of-cans
            (inventory-number-of-cans requested-drink inventory)))
      (<= requested-number-of-cans number-of-cans))))

```

`Order-satisfiable?` benutzt die Prozedur `inventory-number-of-cans`, um zu ermitteln, wieviele Dosen einer bestimmten Getränkesorte sich noch im Bestand befinden. Die Prozedur ruft dazu die praktische Higher-Order-Hilfsprozedur `any` auf, die in einer Liste ein Element findet, das ein bestimmtes Prädikat erfüllt; dieses Element wird zurückgegeben:

```

(define inventory-number-of-cans
  (lambda (drink inventory)
    (cdr
     (any (lambda (inventory-entry)
            (equal? drink (car inventory-entry)))
          inventory))))

(define any
  (lambda (proc? list)
    (if (null? list)
        #f
        (if (proc? (car list))
            (car list)
            (any proc? (cdr list))))))

```

Die Prozedur `order-price` berechnet den Preis einer Bestellung. Zur Erinnerung: eine Bestellung ist ein Paar aus dem Namen einer Getränkesorte und der Anzahl der bestellten Dosen. `order-price` muß dazu den Preis einer Dose aus der Preiliste ermitteln und ruft dazu die Prozedur `drink-price` auf:

```

(define order-price
  (lambda (order prices)
    (let* ((drink (car order))
           (price (drink-price drink prices))
           (number-of-cans (cdr order)))
      (* number-of-cans price))))

(define drink-price
  (lambda (drink prices)
    (if (null? prices)
        (error "unfound drink")
        (if (equal? drink (car (car prices)))
            (cdr (car prices))
            (drink-price drink (cdr prices))))))

```

Einige Hilfsprozeduren sind für die Bestätigung einer Bestellung, die Entgegennahme der Bezahlung und die Anzeige des Wechselgelds zuständig:

```

(define confirm-order
  (lambda (order)
    (display "You ordered ")
    (display (cdr order))
    (display " cans of ")
    (display (car order))
    (display ".")
    (newline)))

(define accept-payment
  (lambda (price)
    (display "Insert ")
    (prompt price)))

(define display-change
  (lambda (price payment)
    (display "Here is your change: ")

```

```
(display (- payment price))
(newline))
```

Wenn eine Bestellung nicht erfüllbar ist, also der Bestand nicht ausreicht, ruft `drink-machine` die Prozedur `display-rejection` auf. Hier ist sie:

```
(define display-rejection
  (lambda (order inventory)
    (display "Sorry, we don't have enough ")
    (display (car order))
    (display " in stock.")
    (newline)))
```

Bleibt zu guter letzt noch `apply-order`: diese Prozedur muß eine Bestellung auf die Bestandsliste anwenden, also beim entsprechenden Eintrag die Dosenanzahl der Bestellung vom Bestand abziehen, und den Rest der Bestandsliste unverändert lassen.

```
(define apply-order
  (lambda (order inventory)
    (if (null? inventory)
        '()
        (let ((inventory-entry (car inventory)))
          (if (equal? (car order) (car inventory-entry))
              (cons (cons (car order)
                          (- (cdr inventory-entry) (cdr order)))
                    (cdr inventory))
              (cons inventory-entry
                    (apply-order order (cdr inventory))))))))))
```

5.5 Repräsentationswechsel

Der Entwicklungsprozeß dieses Programms ist anfällig für Fehler. Das liegt daran, daß es die gleiche Repräsentation für verschiedene Sorten verwendet. Insbesondere tauchen an mehreren Stellen Paare aus jeweils einem Symbol und einer Zahl auf. Ist nun ein Wert

```
(coke . 20)
```

eine Preisangabe für Cola oder aber der Bestand an Cola im Automaten? Von außen läßt sich das dem Wert nicht ansehen. Merkwürdige Effekte entstehen, wenn zum Beispiel beim zweiten rekursiven Aufruf von `drink-machine` die Argumente `prices` und `inventory` vertauscht werden:

```
...
(begin
  (display-rejection order inventory)
  (drink-machine inventory prices))))))
```

Es könnte nun folgender Dialog entstehen:

```
> (drink-machine prices inventory)
The inventory:
drink: coke stock: 300
drink: fanta stock: 100
drink: karamalz stock: 200
```

```

drink: black-bush stock: 120
What drink?
black-bush
How many cans?
200
Sorry, we don't have enough black-bush in stock
The inventory:
drink: coke stock: 100
drink: fanta stock: 100
drink: karamalz stock: 120
drink: black-bush stock: 250
What drink?

```

Es ist noch nicht einmal eine Fehlermeldung erschienen — der Automat hat nur spontan seinen Bestand gewechselt.

Solche Fehler sind in dem Programm schwer zu finden, weil die Repräsentationen für das Inventar und die Preisliste nicht voneinander zu unterscheiden ist. Das Finden läßt sich dramatisch erleichtern, indem die Repräsentationen unterschiedlich gemacht werden. So ist es zum Beispiel möglich, die Preisliste durch eine Prozedur zu repräsentieren. Sie nimmt als Parameter den Namen eines Getränks und gibt den Preis zurück:

```

(define prices
  (lambda (drink)
    (if (equal? drink 'coke)
        100
        (if (equal? drink 'fanta)
            100
            (if (equal? drink 'karamalz)
                120
                (if (equal? drink 'black-bush)
                    250
                    (error "unfound drink"))))))))

```

Die Prozedur `drink-price` im alten Programm hat Gebrauch von der Tatsache gemacht, daß die Preisliste eine Liste von Paaren ist. Sie muß ersetzt werden:

```

(define drink-price
  (lambda (drink prices)
    (prices drink)))

```

Nun funktioniert das Programm mit den Änderungen wieder wie gehabt. Werden allerdings die Parameter vertauscht wie oben, erscheint nun eine Fehlermeldung:

```

for-each: expects type <list> as 2nd argument, given:
#<procedure:prices>; other arguments were: #<procedure>

```

Damit läßt sich schon deutlich mehr anfangen. Vor allen Dingen hat das Scheme-System bemerkt, daß etwas nicht in Ordnung ist.

Auf ähnliche Art und Weise läßt sich auch die Repräsentation von `inventory` verändern. Dort allerdings gibt es mehr Prozeduren, die „wissen“, was die Repräsentation ist: `inventory-number-of-cans`, `display-inventory`, `empty?` und `apply-order`.

Die neue Repräsentation für `prices` versteckt ihre Innereien: eine Prozedur läßt kaum Information nach außen darüber dringen, was sie eigentlich enthält und tut. Dies verringert die Wahrscheinlichkeit, daß fehlerhafte (oder boshafte) Programmteile die Repräsentation manipulieren können, ohne „Insider-Wissen“ zu haben.

Die Repräsentationsänderung für `prices` hat sich nur auf eine einzelne Prozedur beschränkt, der Rest des Programms bleibt unverändert. Das ist kein Zufall: Das Prinzip der abstrakten Datentypen ist es, einen Datentyp über das Verhalten seiner Operationen zu schreiben. Daraus folgen die drei obersten Prinzipien der Datenabstraktion:

1. Für eine neue Sorte Wert, stelle Operationen zur Verfügung, welche es erlauben, die Werte des Typs manipulieren, die aber nicht dessen konkrete Repräsentation „verraten“.
2. Benutze für die Manipulation der Werte der neuen Sorte *ausschließlich* die neuen Operationen, nicht aber mehr die Operationen, die für die Repräsentation verwendet wurden.
3. Verhindere so gut als möglich den Zugriff auf die Werte durch die Wahl einer Repräsentation, welche den Zugriff über andere Operationen als die definierten verwehrt.

5.6 Abstraktionen für Datenabstraktion

Auf Dauer ist es unpraktisch, sich für jeden einzelnen neuen Datentyp, der mit einem anderen verwechselt werden könne, sich eine neue Repräsentation einfallen zu lassen. Die Erzeugung neuer Repräsentationen läßt sich zum Glück automatisieren.

Zu diesem Zweck werden die Datenwerte in *Typen* eingeteilt. Jeder Typ bekommt eine zugeteilte Repräsentation, die alle Werte des Typs verwenden. In diese Repräsentation wird die eigentliche, vom Programm benutzte Repräsentation, eingepackt. Ein Typ wird erzeugt aus einem Namen für den Typ, z.B. also:

```
(define price-type (make-type 'price))
```

Die `make-type`-Prozedur verwendet für die Repräsentation des Typen zunächst einmal dessen Namen. (Die `make-type`-Prozedur mag etwas überflüssig erscheinen. Es wird allerdings in einem der folgenden Kapitel opportun werden, ihre Definition zu ändern.)

```
(define make-type
  (lambda (name)
    name))
```

Zu jedem Typ gehört nun ein sogenannter *Konstruktor*, der Werte dieses Typs erzeugt.

```
(define typed-value-maker
  (lambda (type)
    (lambda (value)
      (cons type value))))
```

Des weiteren gehört ein *Selektor* dazu, der den ursprünglichen Wert aus dem getypten Wert extrahiert:

```
(define typed-value-selector
  (lambda (type)
    (lambda (value)
      (if (equal? type (car value))
          (cdr value)
          (error "type mismatch")))))
```

Manchmal ist außerdem ein *Prädikat* praktisch, das feststellt, ob ein Wert zum Typ gehört oder nicht:

```
(define typed-value-predicate
  (lambda (type)
    (lambda (value)
      (and (pair? value)
           (equal? type (car value))))))
```

Mit der Hilfe dieses Mechanismus lassen sich Preislisten und Inventarlisten disjunkt repräsentieren:

```
(define prices-type (make-type 'prices))
(define make-prices (typed-value-maker prices-type))
(define prices-alist (typed-value-selector prices-type))
```

```
(define drink-price
  (lambda (drink prices)
    (drink-price-1 drink (prices-alist prices))))
```

```
(define drink-price-1
  (lambda (drink alist)
    (if (null? alist)
        (error "unfound drink")
        (if (equal? drink (car (car alist)))
            (cdr (car alist))
            (drink-price-1 drink (cdr alist))))))
```

```
(define inventory-type (make-type 'inventory))
(define make-inventory (typed-value-maker inventory-type))
(define inventory-alist (typed-value-selector inventory-type))
```

```
(define inventory-number-of-cans
  (lambda (drink inventory)
    (cdr
     (any (lambda (inventory-entry)
           (equal? drink (car inventory-entry)))
         (inventory-alist inventory)))))
```

```
(define display-inventory
  (lambda (inventory)
    (display "The inventory: ")
    (newline)
    (for-each (lambda (inventory-entry)
                (display "drink: ")
                (display (car inventory-entry))
                (display " stock: ")
                (display (cdr inventory-entry))
                (newline))
              (inventory-alist inventory))))
```

```
(define empty?
  (lambda (inventory)
    (every? (lambda (inventory-entry)
              (= 0 (cdr inventory-entry)))
```

```

(inventory-alist inventory))))

(define apply-order
  (lambda (order inventory)
    (make-inventory
     (apply-order-1 order (inventory-alist inventory)))))

(define apply-order-1
  (lambda (order alist)
    (if (null? alist)
        '()
        (let ((alist-entry (car alist)))
          (if (equal? (car order) (car alist-entry))
              (cons (cons (car order)
                          (- (cdr alist-entry) (cdr order)))
                    (cdr alist))
              (cons alist-entry
                    (apply-order-1 order (cdr alist))))))))))

```

Der Getränkeautomat mit vertauschten Parametern ist dadurch zwar immer noch nicht korrekt, aber verweigert ebenfalls die Arbeit:

```

> (drink-machine prices inventory)
The inventory:
drink: coke stock: 300
drink: fanta stock: 100
drink: karamalz stock: 200
drink: black-bush stock: 100
What drink?
black-bush
How many cans?
120
Sorry, we don't have enough black-bush in stock.
The inventory:
type mismatch

```

Bei `typed-value-maker` und Freunden fällt auf, daß sie lediglich verhindern, daß Werte verschiedener Typen durcheinanderkommen. Die Repräsentationsfrage wird von `typed-value-maker` gar nicht berührt und muß nach wie vor „von Hand“ durchgeführt werden.

Die meisten Programmiersprachen haben eingebaute Mittel für die Unterscheidung der Werte verschiedener Typen und für die Repräsentation zusammengesetzter Werte, häufig in einem einzelnen Mechanismus vermischt. Beispiele sind Objekte in Java oder Records in C. Diese Programmiersprachen führen zusätzliche Typüberprüfungen bei der Übersetzung durch, die es Programmierern erlaubt, Typfehler bereits vor dem ersten Programmdurchlauf zu finden.