

Kapitel 9

Objektorientiertes Programmieren

Durch die Einführung von Zuweisungen hat eine wesentliche Paradigmenverschiebung stattgefunden, was die Konzepte des Programmierens und die Möglichkeiten der Modellierung von Problemen betrifft. Zusammenfassen läßt sich diese Verschiebung folgendermaßen:

Funktionale Modelle sind die Modelle, die vor der Einführung der Zuweisungen ausschließlich benutzt wurden.

- Variablen stehen für Werte.
- Datenstrukturen sind durch ihre Zusammensetzung und ihre Einzelteile bestimmt: Zwei Strukturen gleicher Form mit den gleichen Bestandteilen sind gleich. Gleichheit läßt sich durch `equal?` testen.

Objektmodelle sind Modelle, die Zustand als Mittel der Modellierung benutzen.

- Variablen sind an Orte gebunden, deren Inhalt sich über die Zeit ändern kann.
- Datenstrukturen haben zusätzlich zur ihrer bloßen Zusammensetzung auch noch eine Identität. Gleichheit läßt sich durch `eq?` testen.

Objektmodelle lassen sich (wie funktionale Programmierung auch) auf vielfältige Art und Weise organisieren. Eine besonders populäre Art der Organisation von Objektmodellen ist die *objektorientierte Programmierung* (kurz *OOP* genannt), um die es in diesem Kapitel geht¹.

9.1 Prozeduren mit variabler Argumentanzahl

Es ist einmal wieder an der Zeit, ein neues Sprachelement von Scheme kennenzulernen. Dieses hat unmittelbar nichts mit objektorientierter Programmierung zu tun,

¹Leider ist „objektorientierte Programmierung“ kein feststehender Begriff. Für jede Definition von objektorientierter Programmierung läßt sich eine zweite finden, die mit der ersten keine erkennbaren Gemeinsamkeiten aufweist. Das gilt auch für die sogenannten „objektorientierten Programmiersprachen“: weder erfordert objektorientiertes Programmierung die Benutzung einer objektorientierten Programmiersprache, noch gibt es einen Kriterienkatalog, der objektorientierte Programmiersprachen von anderen zuverlässig unterscheiden könnte. Insbesondere gibt es objektorientierte Programmiersprachen, die auf keinem Objektmodell basieren. Darum beschäftigt sich dieses Kapitel nur mit einer Variante des objektorientierten Programmierens, allerdings mit der weitem populärsten.

wird aber dabei behilflich sein. Alle bisherigen Prozeduren, die mit `lambda` erzeugt wurden, haben eine feste Anzahl von Parametern. Einige der eingebauten Prozeduren jedoch erlauben die Anwendung auf eine variable Anzahl von Argumenten:

```
> (list 1 2)
(1 2)
> (list 1 2 3)
(1 2 3)
```

Solche Prozeduren lassen sich auch in Programmen erzeugen: Die Parameterfolge nach dem `lambda` kann vor dem letzten Parameter einen Punkt aufweisen, etwa so:

```
(lambda (p1 ... pn-1 . pn) b)
```

Wenn eine solche Prozedur aufgerufen wird, so werden die ersten $n - 1$ Argumente an $p_1 \dots p_{n-1}$ gebunden. Alle übrigen Argumente werden in eine Liste verpackt, die an p_n gebunden wird. Hier ein Beispiel:

```
(define f
  (lambda (x . y)
    y))
> (f 1 2 3 4)
(2 3 4)
```

Diese Form von `lambda`-Ausdrücken bedingt, daß es zumindest einen Parameter vor dem Punkt gibt. Die eingebaute Prozedur `list` allerdings (die ebenfalls eine variable Anzahl von Argumenten akzeptiert) kommt sogar ganz ohne Argumente aus:

```
> (list)
()
```

Falls also *alle* Argumente in einer Liste verpackt werden sollen, so wird `lambda` nicht nur ohne Punkt, sondern sogar ohne Klammern eingesetzt:

```
(define g
  (lambda x
    x))
> (g 1 2 3 4)
(1 2 3 4)
```

In der Tat verhält sich `g` ebenso wie `list`.

In der Scheme-Grammatik kommen also zur Regel von `<formals>` noch zwei Fälle hinzu.

```
<lambda expression> ::= (lambda <formals> <body>)
<formals> ::= (<variable>*)
           | <variable>
           | (<variable>+ . <variable>)
```

Diese Form von `lambda` macht also aus einer Folge von Argumenten eine normale Liste.

Manchmal ist es praktisch, diesen Prozeß umkehren zu können, also aus einer normalen Liste eine Argumentfolge zu machen. Die eingebaute Prozedur `apply` akzeptiert zwei Parameter: eine Prozedur und eine Liste. `Apply` wendet die Prozedur auf die Elemente der Liste an:

```
> (apply + '(1 2 3 4))
10
```

9.2 OOP = MPS + Zustand + self + Vererbung

Objektorientiertes Programmieren entsteht durch die Kombination der altbekannten Techniken Message-Passing Style, Zuweisungen sowie den neuen Techniken Selbstbezug und *Vererbung*.

Im Rahmen dieses Kapitels ist ein *Objekt* eine Prozedur, die eine *Nachricht* als Parameter akzeptiert und, abhängig von der Nachricht, eine weitere Prozedur zurückgibt — eine sogenannte *Methode*:

```
(define get-method
  (lambda (object message)
    (object message)))
```

Hier ist ein Konstruktor für eine einfache Sorte Objekt — einen *Redner*, der Dinge sagen kann. Manchmal sagen Redner unangenehme Dinge. In diesem Fall ist es erlaubt, ihnen eine herunterzuhauen. Wenn das dreimal passiert ist, tut es ihnen weh:

```
(define make-speaker
  (lambda ()
    (let ((slaps 0))
      (letrec
        ((self
          (lambda (message)
            (cond ((eq? message 'say)
                  (lambda (stuff)
                    (display stuff)
                    (newline)))
                  ((eq? message 'slap)
                  (lambda ()
                    (set! slaps (+ 1 slaps))
                    (if (= slaps 3)
                        (begin
                          ((self 'say) '(ouch!))
                          (set! slaps 0))))
                    (else (make-no-method 'speaker))))))
          self))))))
```

In der Definition von `make-speaker` ist eine Vorkehrung getroffen worden, damit ein Redner auf sich selbst über die Variable `self` zugreifen kann. Dies ist notwendig für die Methode `slap`, die nach Erreichen der Schmerzgrenze (`ouch!`) sagt, indem sie dem Redner selbst eine `say`-Nachricht schickt. Die `slaps`-Variable führt Buch über den Schmerzpegel, also den inneren Zustand eines Redners.

`Make-speaker` benutzt einen Hilfskonstruktor `no-method`, dessen Rückgabewert sich von einer Methode unterscheiden läßt. Dazu werden ein weiteres Mal die Abstraktionen für Typen benutzt:

```
(define no-method-type (make-type 'no-method))
```

```
(define make-no-method (typed-value-maker no-method-type))
```

```
(define no-method? (typed-value-predicate no-method-type))
```

```
(define no-method-message (typed-value-selector no-method-type))
```

Im Rahmen des Systems für objektorientierte Programmierung wird eine Methode dadurch erkannt, daß getestet wird, daß es sich nicht um einen `no-method`-Wert handelt:

```
(define method?
  (lambda (x)
    (not (no-method? x))))
```

Das direkt Aufrufen von Methoden erfordert inordinat viele Klammern:

```
> ((george 'say) '(the sky is blue))
(the sky is blue)
```

Aber mit Prozeduren mit variabler Anzahl von Parametern läßt sich das Problem lösen:

```
(define send
  (lambda (object message . args)
    (let ((method (get-method object message)))
      (if (method? method)
          (apply method args)
          (error "No method" message (no-method-message method))))))
```

Die Einführung der `send`-Prozedur hat noch einen anderen Zweck — sie muß noch einmal geändert werden — doch dazu später.

Nun sieht der Methodenaufruf deutlich natürlicher aus:

```
> (send george 'say '(the sky is blue))
(the sky is blue)
```

Ein *Dozent* könnte auch eine Art Redner sein, aber einer, der eine zusätzliche Methode `lecture` zum Dozieren hat:

```
(define make-lecturer
  (lambda ()
    (let ((speaker (make-speaker)))
      (letrec
        ((self
          (lambda (message)
            (cond
              ((eq? message 'lecture)
               (lambda (stuff)
                 (send self 'say stuff)
                 (send self 'say '(abstraction abstraction abstraction))))
              (else
               (get-method speaker message))))))
         self))))))
```

Ein Dozent hat also eine Methode `say`, die genauso funktioniert wie bei normalen Rednern, aber auch eine Methode `lecture`:

```
(define mike (make-lecturer))
> (send mike 'say '(the sky is blue))
(the sky is blue)
> (send mike 'lecture '(the sky is blue))
(the sky is blue)
(abstraction abstraction abstraction)
```

Dieses Prinzip — die Erzeugung eines Objekts, das die Eigenschaften eines anderen *und noch zusätzliche* hat — heißt *Vererbung*. Ein Konstruktor für solche Objekte

heißt in diesem Zusammenhang *Klasse*². Die `speaker`-Klasse ist in diesem Zusammenhang eine *Oberklasse* von `lecturer`; `lecturer` ist die *Unterklasse* von `speaker`. Ein Objekt, das von einer Klasse erzeugt wurde, heißt auch *Instanz* dieser Klasse. Lokale Variablen einer Instanz — wie zum Beispiel `slaps` in `make-speaker` — heißen *Instanzvariablen*.

9.3 Vererbung und self

Das Prinzip der Vererbung läßt sich weitertreiben. Ein `arrogant-lecturer` könne ein Dozent sein, der allem, was er sagt, „it is obvious that“ vorausschickt. Dazu muß die `say`-Methode von des `lecturer`-Objekts durch eine andere ersetzt, diese also *überschrieben* werden:

```
(define make-arrogant-lecturer
  (lambda ()
    (let ((lecturer (make-lecturer)))
      (letrec
        ((self
          (lambda (message)
            (cond ((eq? message 'say)
                  (lambda (stuff)
                    (send lecturer 'say
                              (append '(it is obvious that)
                                      stuff))))
                 (else (get-method lecturer message))))))
         self))))))
```

In der Tat funktioniert der Präfix, wenn die Methode `say` benutzt wird:

```
(define dick (make-arrogant-lecturer))
> (send dick 'say '(the sky is blue))
(it is obvious that the sky is blue)
```

Die Arroganz verschwindet aber auf merkwürdige Art und Weise im Vorlesungssaal:

```
> (send dick 'lecture '(the sky is blue))
(the sky is blue)
(abstraction abstraction abstraction)
```

Das ist wahrscheinlich nicht im Sinne des Erfinders. Der Grund liegt darin, daß die `lecture`-Methode aus dem `lecturer`-Objekt übernommen wird. Diese wiederum beauftragt `self` mit der Aussprache. `Self` bezeichnet aber den `lecturer`, nicht `dick`, und damit gibt es auch keinen „Obvious“-Präfix.

Das grundlegende Problem ist also, daß die Methoden von `lecturer` mit dem falschen Wert für `self` hantieren: Damit der richtige Wert benutzt wird, muß dieser bei Methodenaufrufen mit übergeben werden. Bei `make-speaker` ist das noch unspektakulär:

```
(define make-speaker
  (lambda ()
    (let ((slaps 0))
```

²In vielen objektorientierten Sprachen ist der Begriff „Konstruktor“ anderweitig vergeben, nämlich für spezielle Methoden, die direkt nach der Erzeugung aufgerufen werden. In diesen Sprachen werden die eigentlichen Konstruktoren — also der Code, der Speicher für ein Objekt reserviert und die Felder anfänglich belegt — automatisch erzeugt. Die Klassen fungieren dort als Muster für die Generierung der Konstruktoren

```

(letrec
  ((self
    (lambda (message)
      (cond ((eq? message 'say)
             (lambda (self stuff)
               (display stuff)
               (newline)))
            ((eq? message 'slap)
             (lambda (self)
               (set! slaps (+ 1 slaps))
               (if (= slaps 3)
                   (begin
                     (send self 'say '(ouch!))
                     (set! slaps 0))))
              (else (make-no-method 'speaker))))))
    self))))

```

Tatsächlich ist damit das `letrec` überflüssig geworden, und `make-speaker` läßt sich vereinfachen:

```

(define make-speaker
  (lambda ()
    (let ((slaps 0))
      (lambda (message)
        (cond ((eq? message 'say)
               (lambda (self stuff)
                 (display stuff)
                 (newline)))
              ((eq? message 'slap)
               (lambda (self)
                 (set! slaps (+ 1 slaps))
                 (if (= slaps 3)
                     (begin
                       (send self 'say '(ouch!))
                       (set! slaps 0))))
                  (else (make-no-method 'speaker))))))
    )))

```

Damit Methodenaufrufe weiter funktionieren wie bisher, muß `send` geändert werden:

```

(define send
  (lambda (object message . args)
    (let ((method (get-method object message)))
      (if (method? method)
          (apply method (cons object args))
          (error "No method" message (no-method-message method)))))

```

Die Änderungen an `make-lecturer` und `make-arrogant-lecturer` sind minutiös, kurieren aber das Problem:

```

(define make-lecturer
  (lambda ()
    (let ((speaker (make-speaker)))
      (lambda (message)
        (cond
          ((eq? message 'lecture)
           (lambda (self stuff)

```

```

      (send self 'say stuff)
      (send self 'say '(abstraction abstraction abstraction))))
    (else
      (get-method speaker message))))))

(define make-arrogant-lecturer
  (lambda ()
    (let ((lecturer (make-lecturer)))
      (lambda (message)
        (cond ((eq? message 'say)
              (lambda (self stuff)
                (send lecturer 'say
                  (append '(it is obvious that)
                          stuff))))
              (else (get-method lecturer message)))))))

```

Nun funktioniert das ganze richtig:

```

(define dick (make-arrogant-lecturer))
> (send dick 'lecture '(the sky is blue))
(it is obvious that the sky is blue)
(it is obvious that abstraction abstraction abstraction)
> (send dick 'slap)
> (send dick 'slap)
> (send dick 'slap)
(it is obvious that ouch!)

```

9.4 Mehrfachvererbung

Für ein Objekt ist es prinzipiell sogar möglich, von mehreren anderen Objekten zu erben. Für das entsprechende Beispiel wird ein *Sänger* bzw. *Sängerin* benötigt, die singen kann aber auch ganz normal sprechen:

```

(define make-singer
  (lambda ()
    (lambda (message)
      (cond
        ((eq? message 'say)
         (lambda (self stuff)
           (display (append '(tra-la-la) stuff))
           (newline)))
        ((eq? message 'sing)
         (lambda (self)
           (display '(tra-la-la))
           (newline)))
        (else (make-no-method 'singer))))))

```

Hier einige Methodenaufrufe:

```

(define heino (make-singer))
> (send heino 'sing)
(tra-la-la)
> (send heino 'say '(the hazelnut is brown))
(tra-la-la the hazelnut is brown)

```

Madonna ist im wesentlichen eine Sängerin aber hält auch manchmal kluge Reden:

```
(define madonna
  (let ((singer (make-singer))
        (lecturer (make-lecturer)))
    (lambda (message)
      (let ((singer-method (get-method singer message))
            (if (method? singer-method)
                singer-method
                (get-method lecturer message)))))))
```

Das `if` am Ende bewirkt daß, wenn Madonna um eine Methode gebeten wird, sie zunächst nachschaut, ob die Sängerin in ihr diese Methode besitzt. Wenn ja, wird sie zurückgegeben, wenn nein die entsprechende Methode der Dozentin in ihr:

```
> (send madonna 'sing)
(tra-la-la)
> (send madonna 'lecture '(the sky is blue))
(tra-la-la the sky is blue)
(tra-la-la abstraction abstraction abstraction)
```

Damit erbt `madonna` gleich von zwei anderen Objekten, je nach Bedarf. Diese Technik heißt *Mehrfachvererbung*. Sie erlaubt mächtige Programmiertricks, erfordert aber Sorgfalt bei der Strategie der Methodenauswahl — wenn mehrere Oberklassen die gleiche Nachricht verstehen, muß eine Vorrang bekommen, wie in diesem Fall grundsätzlich die der `singer`-Klasse.

Politiker müssen Vorträge halten können aber auch die Nationalhymne singen:

```
(define norbert
  (let ((singer (make-singer))
        (lecturer (make-lecturer)))
    (lambda (message)
      (let ((lecturer-method (get-method lecturer message))
            (if (method? lecturer-method)
                lecturer-method
                (get-method singer message)))))))
```

Im Zweifelsfall aber hält Norbert lieber einen trockenen Vortrag:

```
> (send norbert 'lecture '(the sky is blue))
(the sky is blue)
(abstraction abstraction abstraction)
```