

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 eine Identität abseits von ihrer Zusammensetzung. 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 geht.

Anmerkung: Leider 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 Programmieren 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.

9.1 Prozeduren mit variabler Parameteranzahl

Es ist einmal wieder an der Zeit, ein neues Sprachelement von Scheme kennenzulernen. Dieses hat unmittelbar nichts mit objektorientierter Programmierung zu tun, wird aber dabei behilflich sein. Alle bisherigen Prozeduren, die mit `lambda` erzeugt wurden, haben eine feste Anzahl von Parametern. Einige der eingebauten Prozeduren jedoch haben eine variable Anzahl von Parametern:

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

Solche Prozeduren lassen sich auch mit `lambda` erzeugen: Wenn die Parameterliste nach dem `lambda` vor dem letzten Parameter einen Punkt aufweist, werden, wenn die Prozedur aufgerufen wird, die Parameter vor dem Punkt an die Werte der ersten Operanden gebunden, alle übrigen aber in einer Liste verpackt an den Parameter nach dem Punkt:

```
(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 Parametern akzeptiert) kommt sogar ganz ohne Operanden aus:

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

Falls also die Werte *aller* Operanden 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 Parameterliste eine normale Liste. Manchmal ist es praktisch, diesen Prozeß umkehren zu können, also aus einer normalen Liste eine Parameterliste 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 + set! + self + x

Objektorientiertes Programmieren entsteht durch die geschickte Kombination von Message-Passing Style, Zuweisungen, Selbstbezug und einem Konzept namens *Vererbung*. Die Vererbung sei zunächst zu einem späteren Abschnitt delegiert — eine Weile lang geht es auch ohne. Die Beispiele dieses Kapitels kommen außerdem auch ohne `set!` aus.

Im Rahmen dieses Kapitels ist ein *Objekt* also 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 einfaches Objekt — ein *Redner* — das Dinge sagen kann:

```
(define make-speaker
  (lambda ()
    (letrec
      ((self
        (lambda (message)
          (cond ((eq? message 'say)
                 (lambda (stuff) (display stuff)))
                (else (no-method 'speaker))))))
      self)))
```

In der Definition von `make-speaker` sind bereits Vorkehrungen getroffen worden, damit `make-speaker` auf sich selbst über die Variable `self` zugreifen kann. `Self` wird in diesem Beispiel noch nicht benötigt, aber das wird sich noch ändern. `Make-speaker` benutzt einen Hilfskonstruktor `no-method`, dessen Rückgabewert sich von einer Methode unterscheiden läßt:

```
(define no-method
  (lambda (name)
    (list 'no-method name)))

(define no-method?
  (lambda (x)
    (if (pair? x)
        (eq? (car x) 'no-method)
        #f)))

(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 ask
  (lambda (object message . args)
```

```
(let ((method (get-method object message)))
  (if (method? method)
      (apply method args)
      (error "No method" message (cadr method))))))
```

(Die Einführung von `ask` hat noch einen anderen Zweck, doch dazu später.)

Nun sieht der Methodenaufruf etwas natürlicher aus:

```
> (ask 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)
                 (ask self 'say stuff)
                 (ask 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))
> (ask mike 'say '(the sky is blue))
(the sky is blue)
> (ask 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 *Klasse*.

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:

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

```

      (ask 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))
> (ask 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:

```

> (ask 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 ()
    (letrec
      ((self
        (lambda (message)
          (cond ((eq? message 'say)
                 (lambda (self stuff)
                   (display stuff)
                   (newline)))
                (else (no-method 'speaker))))))
      self)))

```

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

```

(define make-speaker
  (lambda ()
    (lambda (message)
      (cond ((eq? message 'say)
             (lambda (self stuff)
               (display stuff)
               (newline)))
            (else (no-method 'speaker))))))

```

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

```

(define ask
  (lambda (object message . args)
    (let ((method (get-method object message)))
      (if (method? method)
          (apply method (cons object args))
          (error "No method" message (cadr 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)
            (ask self 'say stuff)
            (ask 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)
                 (ask lecturer 'say
                  (append '(it is obvious that)
                          stuff))))
              (else (get-method lecturer message))))))))
```

Nun funktioniert das ganze richtig:

```
(define dick (make-arrogant-lecturer))
> (ask dick 'lecture '(the sky is blue))
(it is obvious that the sky is blue)
(it is obvious that abstraction abstraction abstraction)
```

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 (no-method 'singer))))))
```

Hier einige Methodenaufrufe:

```
> (ask heino 'sing)
(tra-la-la)
> (ask 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 ((sing (get-method singer message))
            (lect (get-method lecturer message)))
        (if (method? sing)
            sing
            lect))))))
```

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:

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

Politiker müssen Vorträge halten können aber auch die Nationalhymne singen:

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

Im Zweifelsfall aber hält Norbert lieber einen trockenen Vortrag:

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