

```
(define add-5 (make-add 5))
> (add-5 7)
12
```

Damit wurde `make-add` benutzt, um eine Prozedur herzustellen, die eine konstante Zahl auf ihr Argument addiert. `Make-add` kam im selben Programm auch noch für die Herstellung weiterer Konstanten-Addier-Prozeduren verwendet werden:

```
(define add-1 (make-add 1))
(define add-23 (make-add 23))
> (add-1 7)
8
> (add-23 7)
30
```

Damit ist `make-add` so etwas wie eine „Prozedurfabrik“: sie kann zur Laufzeit des Programms verwendet werden, eine ganze Familie von Prozeduren herzustellen.

`Make-add` ist bereits eine praktische kleine Prozedur, scheint aber noch nicht besonders geeignet, die Programmierung revolutionär zu effektivieren. Ein typisches Beispiel für eine Higher-Order-Funktion in der Mathematik ist die Funktionskomposition \circ . Seien $f : B \rightarrow C$ und $g : A \rightarrow B$ Funktionen, dann ist $f \circ g$ folgendermaßen definiert:

$$(f \circ g)(x) := f(g(x))$$

o läßt sich direkt in Scheme übertragen:

```
(define compose
  (lambda (f g)
    (lambda (x)
      (f (g x)))))
```

Dabei müssen `f` und `g` jeweils für Prozeduren mit einem Parameter stehen:

```
(define add-28 (compose add-5 add-23))
> (add-28 3)
31
> ((compose (lambda (x) (* x 2)) add-1) 5)
12
```

Das letzte Beispiel demonstriert auch noch einmal die Verwendung einer Prozedur, die nicht vorher durch `define` einen Namen bekommen hat. Solche „anonymen Prozeduren“ sind eigentlich eine ganz natürliche Sache — möglicherweise etwas ungewohnt, da sowohl die Funktionsdefinition in der Mathematik als auch die Definition von Prozeduren in vielen herkömmlichen Programmiersprachen in der Regel die Vergabe eines Namens erfordert.

`Compose` läßt sich benutzen, um eine weitere praktische Higher-Order-Prozedur zu definieren — `repeat`:

```
(define repeat
  (lambda (n proc)
    (if (= n 0)
        (lambda (x) x)
        (compose proc (repeat (- n 1) proc)))))
```

`Repeat` ist dabei das Pendant zur Exponentiation von Funktionen in der Mathematik, wo f^n für eine Funktion $f : A \rightarrow A$ und $n \in \mathbb{N}$ folgendermaßen definiert

Kapitel 4

Higher-Order-Programmierung

Eine ganze Armada äußerst effektiver Programmier-Techniken beruht auf der Verwendung von Prozeduren, die entweder selbst Prozeduren als Argumente haben oder Prozeduren zurückgeben — oder beides. Solche Prozeduren heißen auch *Prozeduren höherer Ordnung* oder *Higher-Order-Prozeduren*. Der Programmierstil, der durch die Verwendung von Higher-Order-Prozeduren entsteht, heißt demnach *Higher-Order-Programmierung*. In Scheme sind diese Higher-Order-Prozeduren eigentlich nichts besonders — schließlich sind Prozeduren Datenwerte wie alle anderen auch. Dennoch erfordert die Anwendung von Higher-Order-Prozeduren etwas Gewöhnung und Training: in der Mathematik tauchen Funktionen höherer Ordnung meist nur in eingeschränkter Form auf. Die meisten populären Programmiersprachen gar unterstützen sie überhaupt nicht — dort sind Prozeduren oder ihre örtlichen Pendants nur eingeschränkt verwendungsfähig: es handelt sich dort um Werte (allerhöchstens) zweiter Klasse.

4.1 Prozedurfabriken

Eine der einfachsten Higher-Order-Prozeduren ist die folgende:

```
(define make-add
  (lambda (a)
    (lambda (b)
      (+ a b))))
```

`Make-add` ist offensichtlich eine Prozedur mit einem Parameter `a`. Weiter innen wird der Wert von `a` auf eine andere Zahl addiert — es muß sich also bei `a` wohl um eine Zahl handeln. Das Substitutionsmodell erklärt, was bei einer Anwendung von `make-add` herauskommt:

```
(make-add 5)
 $\implies$  ((lambda (a) (lambda (b) (+ a b))) 5)
 $\implies$  (lambda (b) (+ 5 b))
```

Herausgekommen ist eine Prozedur, die auf ihr Argument 5 addiert:

```
> ((make-add 5) 7)
12
```

`Make-add` ist also so etwas wie `+`, nur daß für die korrekte Anwendung mehr Klammern notwendig sind als bei `+`. Allerdings ist es gar nicht erforderlich, `make-add` sofort „vollständig“ anzuwenden:

ist:

$$f^n := \underbrace{f \circ \dots \circ f}_{n\text{-mal}}$$

c

4.2 Higher-Order-Prozeduren auf Listen

Viele der praktischsten Prozeduren auf Listen sind Higher-Order-Prozeduren oder lassen sich mit Hilfe von solchen definieren. Das klassische Beispiel ist `map`, eine Prozedur, die als Parameter selbst eine Prozedur und eine Liste hat. `Map` wendet diese Prozedur auf alle Elemente der Liste an und produziert eine Liste der Rückgabewerte:

```
(define map
  (lambda (proc l)
    (if (null? l)
        '()
        (cons (proc (car l))
              (map proc (cdr l))))))
```

`Map` läßt sich beispielsweise folgendermaßen anwenden:

```
> (map add-23 (list 1 2 3 4 5))
(24 25 26 27 28)
```

Die `map`-Prozedur codiert damit ein häufig vorkommendes Rekursionsmuster — wenn eine Programmierin genau eine Liste benötigt, die aus einer anderen durch Anwendung einer Prozedur auf ihre Elemente entsteht, muß sie nicht selbst Rekursion einsetzen, sondern kann einfach `map` benutzen.¹

Die Codierung von Rekursionsmustern als Higher-Order-Prozeduren ist in der Tat eine mächtige Abstraktionstechnik. Allerdings ist `map` noch gar nicht die nützlichste Higher-Order-Prozedur auf Listen. Diesen Platz nimmt eine Prozedur namens `fold-right` ein:

```
(define fold-right
  (lambda (proc unit l)
    (if (null? l)
        unit
        (proc (car l)
              (fold-right proc unit (cdr l))))))
```

`Fold-right` funktioniert wie folgt: Die Prozedur hat als Parameter eine Prozedur mit zwei Parametern, einen Wert, und eine Liste von Werten. Es gilt folgende Gleichung, in welcher der erste Parameter als binäre Operation geschrieben ist:

$$(\text{fold-right } \odot \ u \ (a_1 \ \dots \ a_n)) = a_1 \odot (a_2 \odot (\dots (a_n \odot u) \dots))$$

Damit wird also effektiv \odot zwischen die Elemente der Liste eingefügt. Zum Beispiel kann `fold-right` benutzt werden, um alle Elemente einer Liste zu addieren:

```
> (fold-right + 0 (list 1 2 3 4 5))
```

15

¹ Umgekehrt heißt das, daß in Programmiersprachen, in denen sich `map` nicht definieren läßt, ganze Horden von Schleifen oder Rekursionsmustern stehen, die „irgendwie gleich“ aussehen.

... oder zu multiplizieren:

```
> (fold-right * 1 (list 1 2 3 4 5))
120
```

`Fold-right` kann auch benutzt werden, um die Länge einer Liste auszurechnen:

```
(fold-right (lambda (n result) (+ result 1)) 0 (list 1 2 3 4 5))
```

Auch die Arbeit von `append` läßt sich mit `fold-right` erledigen:

```
> (fold-right cons (list 4 5 6) (list 1 2 3))
(1 2 3 4 5 6)
```

Warum ist `fold-right` so universell einsetzbar? Zur Erinnerung, das allgemeine Rekursionsmuster für Prozeduren über Listen sah folgendermaßen aus:

```
(define p
  (lambda (l)
    (if (null? l)
        ?
        (... (car l) ... (p (cdr l)) ...))))
```

Genau über die Lücken dieses Musters ist aber in `fold-right` abstrahiert. Nahezu alle Prozeduren über Listen lassen sich so mit Hilfe von `fold-right` definieren. Einige praktische Beispiele:

Filter liefert nur diejenigen Elemente einer Liste, für die ein als Argument übergebenes Prädikat `#t` liefert:

```
(define filter
  (lambda (proc? l)
    (fold-right (lambda (first result)
                  (if (proc? first)
                      (cons first result)
                      result))
                '()
                l)))
```

Die Prozedur `every?` findet heraus, ob ein übergebenes Prädikat auf alle Elemente einer Liste zutrifft:

```
(define every?
  (lambda (proc? l)
    (fold-right (lambda (first result)
                  (if result
                      (proc? first)
                      #f))
                '()
                l)))
```

`Every?` führt damit eine Und-Verknüpfung zwischen den Rückgabewerten des Prädikats auf den Elementen der Liste durch. Dies wird bei Verwendung des `if` nicht auf den ersten Blick klar — viel schöner wäre es, wenn dort das Wort `and` signalisieren könnte, was vor sich geht. In der Tat gibt es in Scheme eine Spezialform namens `and`, die ihre booleschen Operanden und-verknüpft. Hier ist die Syntax:

```
(expression) ::= (and (test)*)
```

And-Spezialformen werden grundsätzlich in `ifs` übersetzt; es handelt sich also, wie auch bei `let`, um syntaktischen Zucker. Es gelten folgende Übersetzungsregeln:

```
(and)  → #t
(and t0 t1 ...) → (if t0 (and t1 ...) #f)
```

Mit Hilfe von `and` kann `every?` etwas klarer formuliert werden:

```
(define every?
  (lambda (proc? l)
    (fold-right (lambda (first result)
                  (and result (proc? first)))
                '()
                l)))
```

Analog zu `and` gibt es `or`:

```
(expression) ::= (or (test)*)
```

Für `or` gilt folgende Übersetzung:

```
(or)  → #f
(or t0 t1 ...) → (if t0 t0 (or t1 ...))
```

Des weiteren gibt es noch eine Prozedur `not`, welche aus `#t` macht und umgekehrt.

Genug des Exkurses: `Fold-right` sammelt die Elemente von hinten nach vorn bzw. von rechts nach links auf, entsprechend der „natürlichen“ Rekursionsstruktur über Listen. Das gleiche Spiel läßt sich auch in der anderen Richtung durchführen. Heraus kommt eine Prozedur `fold-left`, die folgende Gleichung erfüllt:

$$(\text{fold-left } \odot u (a_1 \dots a_n)) = (\dots((u \odot a_1) \odot a_2) \dots \odot a_n)$$

Hier ist die Definition von `fold-left`:

```
(define fold-left
  (lambda (proc unit l)
    (fold-left-1 proc unit l unit)))
```

```
(define fold-left-1
  (lambda (proc unit l result)
    (if (null? l)
        result
        (fold-left-1 proc unit (cdr l) (proc result (car l))))))
```

4.3 Der Schönfinkel-Isomorphismus

Noch einmal zurück zum Anfang — dort war von der Prozedur von `make-add` die Rede mit folgender Definition:

```
(define make-add
  (lambda (a)
    (lambda (b)
      (+ a b))))
```

`Make-add` ist eine Art andere Version von `+`, nämlich eine die nicht zwei Argumente auf einmal akzeptiert sondern „nacheinander“. Summen von zwei Zahlen, normalerweise geschrieben als `(+ a b)` lassen sich auch als `(make-add a) b` schreiben. Diese Transformation von einer Prozedur mit zwei Parametern in eine Prozedur mit nur einem Parameter, die eine Prozedur mit einem Parameter zurückgibt, die dann schließlich den „Wert“ liefert läßt sich auch auf andere Prozeduren anwenden:

```
(define make-mult
  (lambda (a)
    (lambda (b)
      (* a b))))
```

```
(define make-cons
  (lambda (a)
    (lambda (b)
      (cons a b))))
```

Erneut folgen eine ganze Familie von Prozeduren einem gemeinsamen Muster, und erneut läßt sich dieses Muster als Prozedur höherer Ordnung formulieren. Die Prozedur `curry` nimmt als Argumente eine Prozedur mit zwei Parametern und liefert eine entsprechend transformierte Prozedur zurück:

```
(define curry
  (lambda (proc)
    (lambda (a)
      (lambda (b)
        (proc a b)))))
```

Nun lassen sich die `make-x`-Prozeduren von oben mit Hilfe von `curry` definieren:

```
(define make-add (curry +))
(define make-mult (curry *))
(define make-cons (curry cons))
```

Die Transformation wurde ursprünglich unabhängig von den Mathematikern Moses Schönfinkel und Haskell Curry entdeckt. Im deutschsprachigen Raum heißt die Transformation darum *schönfinkeln*, im englischsprachigen Raum *curryfy* oder *curryfizieren*.

Die Schönfinkel-Transformation läßt sich offensichtlich auch umdrehen:

```
(define uncurry
  (lambda (proc)
    (lambda (a b)
      ((proc a) b))))
```

Damit ist die Transformation ein *Isomorphismus*; es gilt folgende Gleichung für Prozeduren `p` mit zwei Parametern:

$$(\text{uncurry } (\text{curry } p)) \equiv p$$