

# Kapitel 1

## Was ist Programmierung?

### 1.1 Wie funktioniert Programmieren?

Programmierung ist tatsächlich die Beschäftigung mit *Berechnungsprozessen* (manchmal auch *Informationsverarbeitungsprozesse* genannt). Ein Berechnungsprozess ist zunächst einmal ein abstraktes Wesen, aber es kann sehr reale (und manchmal beängstigende) Auswirkungen auf die Außenwelt haben: ein Berechnungsprozess kann einen Computer dazu bringen so zu tun, als sei er ein Taschenrechner, Text zu verarbeiten, oder ein Kernkraftwerk zu steuern.

Ein Berechnungsprozess verrichtet seine Arbeit, in dem er *Daten* (ein weiteres abstraktes Konzept) manipuliert. Gesteuert wird er von einem *Programm* — eine Ansammlung von Regeln, die genau angeben, wie der Prozess sich verhalten soll. Beeinflusst werden kann der Verlauf eines Prozesses durch Ereignisse in der Außenwelt, zum Beispiel durch Interaktion mit einem Benutzer. All diese Konzepte — Berechnungsprozess, Daten, Programm, Außenwelt — entsprechen nicht wirklich der physikalischen Realität eines handelsüblichen PCs, aber sie vereinfachen es, Programme zu verstehen und zu manipulieren.

### 1.2 Maximen des guten Programmierens

Obi-Wan Kenobi wußte es bereits: Was ist ein Magier wenn nicht ein praktizierender Theoretiker? Der Konflikt zwischen Theorie und Praxis ist ein immerwährendes Thema in der Informatik (und nicht nur dort). Das Herzstück des Konflikts ist die Frage zwischen *Anwendung* und *Anwendbarkeit*. Gute Programmierer wissen, daß es oft notwendig ist, einen Schritt zurückzutreten und sich mit etwas zu beschäftigen, daß zum Verständnis des Wesens der Programmierung beiträgt. Diese „Exkursionen in das Abstrakte“ führen nicht immer sofort zu realen Anwendungen und Umsatz in einem konkreten Projekt, zeigen aber oft langfristig großen Nutzen.

Was also macht gute Programmierung bzw. ein gutes Programm aus?

- Korrektheit
- Kürze
- Verständlichkeit
- Allgemeine Anwendbarkeit
- Eleganz und Schönheit
- Coolness

- Hilft es bei der nächsten Cocktail-Party?
- Sieht es gut auf einem T-Shirt aus?

Wie sich herausstellt, sind Programme, die diese Kriterien erfüllen, zumeist auch äußerst nützlich.

### 1.3 Elemente des Programmierens

Eine wissenschaftliches oder zumindest systematische Vorgehensweise bei der Beantwortung der Frage „Was ist  $X$ ?“ ist, die Bestandteile von  $X$  aufzuzählen. Was also ist ein Programm? Ein Programm ist ein Text in einer speziellen Sprache, einer *Programmiersprache*. Eine Programmiersprache ist eine sehr spezielle und eingeschränkte Sprache verglichen mit einer natürlichen Sprache. Dennoch erlaubt eine Programmiersprache, äußerst komplizierte Ideen auszudrücken.

Wie lassen sich Ideen in natürlichen Sprachen ausdrücken? Die Bestandteile einer natürlichen Sprache sind — grob vereinfacht gesagt — die folgenden:

**Wörter aus dem Wörterbuch** repräsentieren Dinge und Aktivitäten.

**Sätze** setzen Wörter zu größeren Einheiten zusammen.

**Namen** sind ebenfalls Wörter, für die allerdings die Bedeutung wählbar ist anstatt durch ein Wörterbuch festgelegt zu sein.

Programmiersprachen sind da fundamental gar nicht so anders:

**Literale** repräsentieren die kleinsten Dinge, mit denen die Programmiersprache hantiert.

**Kombinationsmittel** machen zusammengesetzte, „größere“ Sprachelemente aus kleineren.

**Abstraktionsmittel** benennen Sprachelemente (meist zusammengesetzte), so daß sie abstrakt und im ganzen manipuliert werden können.

Der Vorgang des Programmierens läuft also grob gesagt folgendermaßen ab:

1. Größere Dinge aus kleineren zusammenbauen.
2. Die größeren Dinge benennen, die Dinge selbst vergessen und nur die Namen im Kopf behalten.
3. Und wieder von vorn.

Der zweite Schritt (eben unter dem Namen *Abstraktion* geläufig) ist dabei das Herzstück der effektiven Programmierung: Sie hilft dem Programmierer, die Anzahl der Dinge zu reduzieren, die er gleichzeitig im Kopf behalten muß, um das Programm weiterzuentwickeln. Dies ist bei den kleinen Programmen der Lehre und Theorie noch nicht so wichtig, wird aber das zentrale Anliegen bei größeren Projekten.

### 1.4 Wie funktioniert Programmieren?

Programmieren läßt sich am besten durch Training lernen. Später wird das Studium guter Programme zu einem weiteren Bestandteil des Prozesses werden.

Dieser Text benutzt für die tatsächlichen Programme die Programmiersprache *Scheme*. Das verwendete Programmiersystem ist *DrScheme*. Es bietet dem Programmierer ein zweigeteiltes Fenster:

1. In der oberen Hälfte des Fensters (dem *Editor*) steht der Programmtext.
2. In der unteren Hälfte des Fensters (der *REPL*) finden die Interaktionen zwischen Benutzer und Programm statt. Außerdem lassen sich hier „Fragen“ an das Programm stellen, um einzelne Programmteile gezielt auszuprobieren.

Ein Scheme-Programm besteht aus einer Aneinanderreihung von sogenannten *Formen*. Manche Formen, die sogenannten *Ausdrücke*, haben ein Ergebnis, einen *Wert*. Beim Druck auf die *Execute*-Taste führt DrScheme den Berechnungsprozeß aus, der zum Programm im Editor gehört. Dabei werden die Werte aller Ausdrücke des Scheme-Programms in der REPL ausgedruckt.

Ein besonders einfaches Programm ist dieses hier:

```
23
```

Dieses Programm besteht aus einem einzelnen Ausdruck, dessen Berechnungsprozeß folgendes Resultat produziert (Überraschung!):

```
23
```

Nun ist das hier eine Universität, also läßt sich aus dieser vermeintlichen Mücke ein verständnistechnischer Elefant machen. Was passiert *wirklich*?

1. Das Programm ist nicht wirklich die Zahl „dreiundzwanzig“, sondern zunächst einmal die Aneinanderreihung der Ziffern 2 und 3.
2. DrScheme übersetzt dieses Programm in Instruktionen für einen Berechnungsprozeß.
3. DrScheme startet den Berechnungsprozeß, dessen Resultat die Zahl „dreiundzwanzig“ ist.
4. DrScheme druckt das Resultat des Berechnungsprozesses in der REPL als die Ziffernfolge 23 aus — die *Repräsentation* der Zahl dreiundzwanzig.

23 ist ein Beispiel für ein Literal. Hier ist eine zusammengesetzte Form:

```
(+ 23 42)
```

Zusammengesetzte Formen heißen in Scheme *Kombinationen*. Sie haben allesamt die gleiche Form:

- eine Klammer auf,
- ein *Operator* der angibt, was zu tun ist,
- 0 bis viele *Operanden* des Operators,
- eine Klammer zu.

(Dies sind in Scheme die *einzig*en Stellen, an denen Klammern auftauchen dürfen. Außerdem dürfen zwischen Kombinationen und zwischen den Teilen einer Kombination beliebig Leerzeichen und Zeilenumbrüche stehen.)

In diesem Fall ist der Operator +, der besagt, daß es um Addition geht. Die Operanden sind die beiden Literale 23 und 42 — die Zahlen, die + addieren soll. Das Resultat ist:

```
65
```

In Scheme gibt es zwei Mechanismen, um Dinge zu benennen. Der einfachere von beiden heißt `define`

```
(define x 23)
```

Dies ist wiederum eine Kombination, aber kein Ausdruck (da sie keinen Wert hat), sondern eine *Definition*. In der REPL läßt sich nach dem Druck auf *Execute* ein Ausdruck eingeben, der zeigt, was die Definition bewirkt hat.

```
> x
23
```

Andere Namen sind möglich:

```
(define karl-otto 423)
(define mehrwertsteuer 16)
(define duftmarke (* 8 4))
```

Der zweite Operand der *define*-Kombination muß ein Ausdruck sein, dessen Wert an den Namen *gebunden* wird.

*Define* ist eine nette Sache. Nur wird seine Nützlichkeit dadurch eingeschränkt, daß es einen Namen nur *einmal* binden kann. Oft wäre es besser, eine etwas allgemeinere Sicht der Benennung zu entwickeln. Will z.B. ein Programmierer den Umfang eines Kreises berechnen, könnte das folgendermaßen vor sich gehen:

```
(define pi 3.14159265)
(define radius 27)
(* 2 pi radius)
```

Was ist, wenn in einem Programm die Umfänge vieler Kreise mit unterschiedlichen Radiussen berechnet werden soll? Das Programm sollte über dem Radius *abstrahieren* und sagen: was immer der Radius sein mag, hier ist eine Regel für die Berechnung des Umfangs. In Scheme heißt eine solche Abstraktion *Prozedur*:

```
(define pi 3.14159265)

((lambda (radius) (* 2 pi radius)) 13)
```

Mit dieser Abstraktion läßt sich auch nicht mehr anfangen als mit dem Programm oben, weil die Abstraktion nur dort verwendet werden kann, wo sie hingeschrieben wurde. Aber *define* erlaubt uns Mehrfachverwendung:

```
(define circumference
  (lambda (radius) (* 2 pi radius)))
```

Jetzt läßt sich in der REPL fragen:

```
> (circumference 13)
81.68140890000001
> (circumference 27)
169.6460031
```

Was ist passiert? Der Ausdruck, der den Umfang berechnet, ist unverändert geblieben. Er befindet sich jetzt nur in einer Kombination, die als Operator *lambda* hat. *Lambda*-Kombinationen heißen auch *Abstraktionen*. Sie sind Ausdrücke, die als Wert ein *Prozedur* haben. Der Ausdruck *(circumference 13)* ist eine *Anwendung* oder ein *Aufruf* der Prozedur, die an *circumference* gebunden ist. Bei einem Aufruf faßt der Berechnungsprozeß sowohl den Operator als auch die Operanden als Ausdrücke auf, die bei der Auswertung zuerst berechnet werden. Der Operator muß als Wert eine Prozedur liefern. Außerdem werden die Werte der Operanden an die Namen nach dem *lambda* gebunden, und die Auswertung fährt mit dem Innern der *Lambda*-Kombination fort.

Ein weiteres Beispiel:

```
(define parking-lot-cars
  (lambda (number-of-vehicles number-of-wheels)
    (/ (- number-of-wheels
          (* 2 number-of-vehicles))
       2)))
```

Dies sind bereits die Elemente der Programmierung.

- Literale
- Abstraktionen
- Anwendungen
- Namen

Alles andere folgt daraus.

Programme hantieren dabei mit Werten, von denen bisher zwei Sorten bekannt sind: Zahlen und Prozeduren.

## 1.5 Programmieren als Problemlösen

Gute Programme lösen Probleme. Im Idealfall ist dabei das Programmieren an sich nicht nur das bloße Eintippen der Lösung, sondern passiert bereits während des Problemlösungsprozesses. Zentral für die Lösung großer Probleme wie sie beim Programmieren auftreten ist das Aufteilen eines größeren Problems. Hier ein — zugegeben etwas akademisches — Beispiel:

```
(define cylinder-volume
  (lambda (radius height)
    (* (* 3.14159265 (* radius radius))
       height)))
```

Im Problem, das zu dieser Prozedur gehört, sind mehrere Unterprobleme versteckt, So ist das Volumen eines Zylinders gerade das Produkt aus Grundfläche und Höhe. Diese Tatsache ist dem obigen Programm nicht unmittelbar anzusehen, lässt sich aber durch die Auslagerung der Grundfläche in eine weitere Prozedur sichtbar machen. Ebenso für die Zahl  $\pi$  und die Quadrierungs-Operation in der Flächenberechnung:

```
(define cylinder-volume
  (lambda (radius height)
    (* (circle-area radius) height)))
```

```
(define pi 3.14159265)
```

```
(define square
  (lambda (x)
    (* x x)))
```

```
(define circle-area
  (lambda (radius)
    (* pi (square radius))))
```

## 1.6 Programme und Berechnungsprozesse

Bisher war es nicht schwer zu erraten, was für ein Ergebnis ein Programm produziert. Wie *genau* sieht der Berechnungsprozeß aus, den ein Programm auslöst? Was „passiert“, wenn DrScheme (`cylinder-volume 5 4`) auswertet? Der Kern einer solchen Erklärung muß sich offensichtlich damit beschäftigen, was bei einem Aufruf passiert. Hier gibt es verschiedene Erklärungsansätze. Ein einfacher Ansatz ist das sogenannte *Substitutionsmodell*.

Die primitivste Tätigkeit, die bei einem Berechnungsprozeß passiert, ist die *Auswertung* einer Form: Der Berechnungsprozeß stellt den Wert einer Form fest.

**Literale** Dies ist einfach für Literale: das Literal `23` hat als Wert die Zahl „dreiundzwanzig.“

**Namen** Für einen Namen wird grundsätzlich der an ihn gebundene Wert ersetzt oder *substituiert*. Namen werden gebunden entweder durch Definitionen oder durch Prozeduraufrufe.

**Abstraktionen** haben als Wert eine Prozedur, deren Bedeutung sich bei der Auswertung von Anwendungen erklärt.

**Anwendungen** erhalten ihren Wert folgendermaßen: Zunächst werden Operator und Operanden als Ausdrücke aufgefaßt und ausgewertet. Dann ist der Wert der Anwendung der Wert des Rumpfes der Prozedur, wobei die Werte der Operanden (die *Parameter*) für die entsprechenden Namen der lambda-Kombination ersetzt werden.

## 1.7 Conditionals und Boolesche Werte

Die bisherigen Programme liefen immer nach demselben Schema ab, nur mit unterschiedlichen Zahlen. Wie steht es mit Funktionen wie dieser hier aus dem Mathematik-Unterricht?

$$|x| := \begin{cases} x & \text{falls } x \geq 0 \\ -x & \text{otherwise} \end{cases}$$

In Scheme sieht die entsprechende Prozedur folgendermaßen aus:

```
(define abs
  (lambda (x)
    (if (>= x 0)
        x
        (- x))))
```

**Abs** benutzt eine neue Kombination, die *if*-Kombination, auch genannt *Conditional*.

Ein *Conditional* hat drei Operanden, allesamt Ausdrücke: den *Test*, die *Konsequente* und die *Alternative*, letztere beide zusammen heißen auch *Zweige*. Abhängig vom Ausgang des Tests ist der Wert des Conditionals entweder der Wert der Konsequente oder der Wert der Alternative:

```
> (>= 3 1)
#t
```

`3` ist größer als `1`, ist der obige Ausdruck, als Aussage aufgefaßt, „wahr“. `#t` steht in der Tat für „true“ oder „wahr“. `>=` ist eine eingebaute Prozedur, welche auf „kleiner oder gleich“ testet.

Andersherum:

```
> (>= 1 3)
#f
```

#f steht für „false“ oder „falsch“. Übrigens gibt es eine andere, äquivalente Schreibweise für abs:

```
(define abs
  (lambda (x)
    ((if (>= x 0) + -) x)))
```

## 1.8 Zusammenfassung

Dieses Kapitel hat sich bis hierher im wesentlichen auf der Ebene von Beispielen bewegt. Hier noch einmal ganz präzise die Form von Scheme-Programmen in Form einer erweiterten BNF. Dabei heißt  $\langle \text{thing} \rangle^*$  „null oder mehr Vorkommen von  $\langle \text{thing} \rangle$ “ und  $\langle \text{thing} \rangle^+$  heißt „ein oder mehr Vorkommen von  $\langle \text{thing} \rangle$ “.

```

<program> ::= <form>*
<form> ::= <definition> | <expression>
<definition> ::= (define <variable> <expression>)
<expression> ::= <variable>
                | <literal>
                | <lambda expression>
                | <conditional>
                | <procedure call>
<lambda expression> ::= (lambda <formals> <body>)
<body> ::= <expression>
<formals> ::= (<variable>*)
<conditional> ::= (if <test> <consequent> <alternate>)
<test> ::= <expression>
<consequent> ::= <expression>
<alternate> ::= <expression>
<procedure call> ::= (<operator> <operand>*)
<operator> ::= <expression>
<operand> ::= <expression>

```

Ein Scheme-Programm manipuliert bei seiner Ausführung *Werte*. Bis dato sind folgende Sorten Werte bekannt:

- Zahlen (23, 17 etc.)
- Prozeduren (als Werte von lambda-Ausdrücken)
- Boolesche Werte, auch genannt *Booleans* (#t und #f)

Diese „Manipulation von Werten“ geschieht durch das Feststellen des Wertes der Ausdrücke eines Programms. Ein Modell für die Feststellung des Wertes eines Ausdrucks ist das *Substitutionsmodell*, das beschreibt, wie ein Ausdruck durch einen anderen ersetzt werden kann, bis Werte dabei herauskommen.

- Ein Name (oder eine Variable) hat als Wert den Wert der rechten Seite der entsprechenden Definition. Bestimmte Namen haben eingebaute Definitionen.
- Ein Literal hat den Wert, für den es steht.

- Eine Abstraktion hat als Wert sich selbst.
- Bei der Auswertung eines Conditionals wird zunächst der Wert des Tests festgestellt. Ist dieser Wert `#t`, so ist der Wert des Conditionals der Wert der Konsequente. Ist er `#f`, so ist der Wert des Conditionals der Wert der Alternative.
- Bei der Auswertung eines Prozeduraufrufs werden zunächst die Werte des Operanden und der Operatoren festgestellt. Der Operator muß eine Prozedur mit ebensoviel Parametern sein, wie der Prozeduraufruf Operanden hat. Der Wert des Prozeduraufrufs ist dabei der Wert des Rumpfes der Prozedur, wobei die Vorkommen der Parameter im Rumpf durch die Werte der Operanden des Ausdrucks ersetzt werden.

Bei der letzten Regel kann es zu scheinbaren Mehrdeutigkeiten kommen. Wie zum Beispiel soll

```
((lambda (x) ((lambda (x) (+ x 1)) (+ x 2))) 13)
```

ausgewertet werden? Das Grundproblem ist dabei, die Vorkommen von `x` in den Rumpfen der Abstraktionen der jeweils richtigen Abstraktion zuzuordnen. Hier gilt das Prinzip der *lexikalischen* Bindung: Ein Name gehört zu der Abstraktion, die ihm, von innen nach außen gesucht, am nächsten liegt. Der obige Ausdruck ist also äquivalent zum folgenden:

```
((lambda (x) ((lambda (y) (+ y 1)) (+ x 2))) 13)
```