

# Kapitel 1

## Was ist Programmierung?

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, indem 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.

### 1.1 Maximen des guten Programmierens

Obi-Wan Kenobi wußte es bereits: Ein Magier ist nichts als ein praktizierender Theoretiker. Der Konflikt zwischen Theorie und Praxis ist ein immerwährendes Thema in der Informatik (und nicht nur dort).

Ist ein Programm, das augenscheinlich funktioniert, auch korrekt? Ist ein augenscheinlich gutes Programm auch immer ein gutes Programm? Ist ein korrektes Programm automatisch ein gutes Programm? Was ist ein gutes Programm?

Diese Fragen lassen sich schwer beantworten, ohne konkrete Programme zu betrachten. Es ist aber nützlich, schon vorher einige Ziele zu formulieren, die über ein einfaches „Es funktioniert doch!“ hinausgehen. An ihnen wird später der Erfolg einer Programmentwicklung messen.

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

- Korrektheit
- Kürze
- Verständlichkeit
- allgemeine Anwendbarkeit
- Erweiterbarkeit
- Eleganz und Schönheit

- Coolness
- gutes Aussehen auf einem T-Shirt
- Tauglichkeit als Konversationsstoff auf einer Cocktail-Party

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

## 1.2 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.

Programmiersprachen bestehen aus folgenden Bestandteilen:

**Literale** sind feste Namen für bestimmte, ausgewählte Datenwerte.

**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 grob gesagt folgendermaßen ab:

1. Größere Dinge aus kleineren zusammenbauen.
2. Die größeren Dinge benennen, die Dinge selbst vergessen und ab dann nur die Namen verwenden.
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.

## 1.3 Wie funktioniert Programmieren?

Programmieren läßt sich am besten durch Training sowie durch das Studium existierender guter Programme lernen.

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
```

Scheinbar trivial, aber intensive akademische Betrachtung erfordert einen Blick hinter die Kulissen. 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 — einen festen Namen für einen Datenwert. Hier ist eine zusammengesetzte Form:

```
(+ 23 42)
```

Zusammengesetzte Formen 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* Stellen, an denen Klammern auftauchen dürfen. Zwischen Formen dürfen 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 zusammengesetzte Form, aber kein Ausdruck, da sie keinen Wert hat. Formen, die keine Ausdrücke sind, heißen *Spezialform*. Eine Spezialform ist stets daran zu erkennen, daß sie als Operator ein bestimmtes Wort hat — ein sogenanntes *syntaktisches Schlüsselwort*. Eine Spezialform, deren syntaktisches Schlüsselwort `define` ist, ist 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 Definition muß ein Ausdruck sein, dessen Wert an den Namen *gebunden* wird. Namen, die nicht Literale sind, heißen *Variablen*.

Definitionen sind nützlich: sie dienen (nicht immer, aber häufig) der Abkürzung. Vor allem aber erlauben sie, für einen für sich betrachtet bedeutungslosen Wert (ist 16 tatsächlich die Mehrwertsteuer, der Spritverbrauch eines Kleinwagens oder das Alter eines Teenagers, in dem er aufhört, die *Bravo* zu lesen?) einen deskriptiven Namen einzuführen und den Wert selbst zu vergessen.

Die Nützlichkeit von Definitionen wird dadurch eingeschränkt, daß es eine Variable 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 Radien 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 daraus. In Scheme heißt der Wert einer solche Abstraktion *Prozedur*:

```
(define pi 3.14159265)

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

Radius ist dabei ein *Parameter* — im obigen Beispiel wird der Wert des Operanden — dreizehn — an den Parameter *radius* gebunden.

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 zusammengesetzten Form, die als Operator *lambda* hat. Lambda-Formen heißen auch *Abstraktionen*: sie abstrahieren über ihre Parameter. Abstraktionen 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 — die sogenannten *Argumente* — an die Parameter der Abstraktion gebunden, und die Auswertung fährt mit dem Innern der Abstraktion fort.

Rückwirkend läßt sich erkennen, daß + und - auch nichts anderes sind als Variablen, die in Scheme vordefiniert sind — sie sind nicht von „normalen“ Prozeduren zu unterscheiden. Dementsprechend sind Ausdrücke wie (+ 23 42) oder (\* 2 pi radius) ganz normale Prozeduraufrufe.

Ein weiteres Beispiel:

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

Damit sind bereits alle Elemente der Programmierung abgedeckt:

- Literale
- Abstraktionen
- Anwendungen
- Variablen

Alles andere folgt daraus.

Bisher sind außerdem zwei Sorten von Datenwerten bekannt: Zahlen und Prozeduren.

## 1.4 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, eine Prozedur zum Errechnen des Volumens eines Zylinders, von dem Radius und Höhe bekannt sind.

```
(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))))
```

Das entstandene Programm ist zwar länger, aber die Bestandteile, aus denen es besteht, sind kürzer geworden: Jede der resultierenden Prozeduren läßt sich (hoffentlich) auf einen Blick verstehen. Als kleiner Bonus lassen sich die Prozeduren `square` und `circle-area` auch unabhängig von `cylinder-volume` verwenden.

## 1.5 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 eines Ausdrucks nach bestimmten Regeln fest. Für jede Sorte Ausdruck gibt es dabei eine eigene Regel.

**Literale** Literale haben stets einen festgelegten Wert, der sich in der Regel direkt ablesen läßt.

**Variablen** Für eine Variable wird grundsätzlich der an sie gebundene Wert ersetzt oder *substituiert*. Variablen 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. Dabei muß der Wert des Operators eine Prozedur sein, die wiederum Wert einer Abstraktion ist, die irgendwo im Programm stehen muß.

Der Wert der Anwendung ist der Wert des Rumpfes der Abstraktion, wobei die Werte der Operanden — die Argumente — für die entsprechenden Parameter der Abstraktion 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*: Eine Variable gehört zu der Abstraktion, die ihr, 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)
```

## 1.6 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 Spezialform, die `if`-Form, 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)))
```

`#t` und `#f` sind gleichzeitig die Literale für „wahr“ und „falsch“:

```
> #t
#t
> #f
#f
```

Zusammenfassend gibt es für die Auswertung von *Conditionals* eine neue Regel im Substitutionsmodell:

**Conditionals** Bei der Auswertung eines *Conditionals* wird zunächst der Wert des Tests festgestellt. Ist dieser Wert „wahr“, so ist der Wert des *Conditionals* der Wert der *Konsequente*. Ist er „false“, so ist der Wert des *Conditionals* der Wert der *Alternative*.

## 1.7 Scheme bis hier auf einen Blick

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> <alternative>)
<test> ::= <expression>
<consequent> ::= <expression>
<alternate> ::= <expression>
<procedure call> ::= (<operator> <operand>*)
<operator> ::= <expression>
<operand> ::= <expression>

```