6. Alternative Programmierkonzepte

6.1 Funktionale Programmierung

6.2 Programmierung mit Logik

6.3 Ereignisgesteuerte Programmierung

6.1 Funktionale Programmierung

Ein Algorithmus wird beschrieben als eine Funktion im mathematischen Sinne, die eine Eingabe auf eine Ausgabe abbildet.

Ein Programm in einer funktionalen Programmiersprache ist abstrakter (problemnäher und maschinenferner) als ein Programm in einer imperativen, sequentiellen Programmiersprache.

Das Prinzip der funktionalen Programmierung (1)

Das gewünschte Resultat eines Programms wird als Funktion seiner Eingabedaten (im mathematischen Sinne) beschrieben. Es können auch Funktionen auf Funktionen verwendet werden.

Alle funktionalen Sprachen beruhen auf dem Lambda-Kalkül von **Church** (1932).

Das Prinzip der funktionalen Programmierung (2)

Ein funktionales Programmiersystem besteht aus

- einer Menge O von Objekten
- einer Menge F von Funktionen, die Objekte auf Objekte abbilden
- einer Operation: Applikation (Anwendung einer Funktion auf ein Objekt)
- einer Menge von Bildungsgesetzen zur Verknüpfung von Funktionen und Objekten (Termbildung)
- einer Menge D von Definitionen, mit deren Hilfe Terme als Funktionen definiert werden und Namen zugeordnet bekommen (Abstraktions-technik zum Einführen komplexer Funktionen)

Einzige Operation ist die Applikation (Anwendung). Geschrieben:

f:x oder f(x)

Semantik der Terme

Ein Term kann sein:

- ein Atom: Ergebnis ist Wert des Atoms.
- eine primitive Funktion: Ergebnis ist der Wert der Applikation auf das Argument
- eine definierte Funktion: Ergebnis erhält man durch Einsetzen des definierten Ausdrucks und Applikation auf das Argument

Wichtigstes und ältestes Beispiel einer funktionalen Sprache: **LISP**

Listen als Basis – Datenstruktur (1)

Viele funktionale Sprachen benutzen **Listen** beliebiger und variabler Länge als Basis - Datentypen, z. B. die Sprache LISP (List Processor).

Beispiel:

```
addiere (x,y) = x + y

Summe (L) = wenn L = [ ] dann 0
            sonst addiere (Kopf (L), Summe (Rumpf (L)))
```

Listen als Basis – Datenstruktur (2)

Die Funktion **Summe** kann jetzt zum Aufsummieren von Listen beliebiger Länge benutzt werden:

```
Summe [3 4 5]= addiere (3, Summe ([4 5]))
             = addiere (3, addiere (4,Summe ([5])))
             = addiere (3, addiere (4,addiere (5,Summe([
               ])))))
             = addiere (3, addiere (4,addiere (5,0)))
             = addiere (3,9)
             = 12
```

Kopf(L): Standardfunktion, liefert erstes Element

Rumpf(L): Standardfunktion, liefert Rest ohne das erste Element

Status der funktionalen Programmiersprachen

Viele funktionale Sprachen wurden entwickelt:

- LISP (Mc Carthy); viele Dialekte
- FP (Backus)
- ML (Gordon)
- Haskell (Hudak und Wadler)
- Scheme
- und viele mehr

Wir werden uns im Folgenden die Sprache ***Scheme*** näher anschauen.

Elemente von SCHEME

Essentielle Mechanismen einer Programmiersprache:

- **Elementare Ausdrücke**: Die einfachsten Einheiten der Sprache.
- **Mittel zur Kombination**: Konstruiere neue Elemente durch Zusammensetzen einfacher Elemente.
- **Mittel zur Abstraktion**: Benenne zusammengesetzte Elemente und behandle/benutze sie als Einheiten.

Elementare Ausdrücke und Prozeduren (1)

Elementare Ausdrücke zur Darstellung von **Zahlen:**

Beispiel:

```
> 15
15
> -3.2
-3.2
```

Ausdruck → Interpreter → Auswertung (Evaluation)

Elementare Ausdrücke und Prozeduren (2)

Nicht alle Ausdrücke stellen Zahlen dar:

Beispiel:

```
> B678
reference to undefined identifier: B678
> -3h
reference to undefined identifier: -3h
```

Elementare Ausdrücke zur Darstellung arithmetischer Prozeduren (**Operationen**):

+, -, *, /

Elementare Ausdrücke und Prozeduren (3)

Anwendung dieser Prozeduren durch Kombination der Ausdrücke **(Präfix-Notation)**:

(Operation Operand Operand {Operand…})

Beispiel:

```
> (+ 17 5)
22
> (+ 17 5 8)
30
> (- 10 3 5 1)
1
> (/ 1 8)
0.125
```

Elementare Ausdrücke und Prozeduren (3)

Die Kombination elementarer Ausdrücke ist ein Ausdruck; die Ausdrücke lassen sich wiederum kombinieren:

Beispiel:

```
> (+ (* 3 5) (/ 48 3))
31
> (- (+ (* 3 5) (/ 48 3)) (* 2 (* 3 5)))
1
```

Anmerkung: Der Wert des Ausdrucks

```
(+ 3 (* 4 5))
```

ist eindeutig, der Wert des aus der Schule bekannten Ausdrucks

```
3 + 4 * 5
```

dagegen ohne weitere Vereinbarung nicht.

Namen und Umgebungen

Für einen Kreis mit Radius r gilt: Umfang: $2\pi r$ Fläche: πr^2

Die Berechnung dieser Ausdrücke für r = 10 ist wie folgt:

```
> (* 2 3.14159 10)
62.8318
> (* 3.14159 (* 10 10))
314.159
```

Das ist wenig leserlich! Es wäre angenehmer, wenn man den Ausdrücken ansehen würde, was sie bzw. ihre Werte bedeuten sollen.

Benennung von Ausdrücken mit `define` (1)

Die Ausdrücke bzw. ihre Werte werden mit **`define`** benannt und dadurch abstrahiert:

```
> (define Pi 3.14159)
> (define radius 10)
> (* 2 Pi radius)
62.8318
> (* Pi (* radius radius))
314.159
> (define kreis-umfang (* 2 Pi radius))
> (define kreis-flaeche (* Pi (* radius radius)))
> kreis-umfang
62.8318
> kreis-flaeche
314.159
```

Benennung von Ausdrücken mit `define` (2)

- Sprechweise: Der **Name** “radius” benennt eine **Variable `radius`**, welche als **Wert** die Zahl 10 hat.
- Die Namen–Wert Zuordnungen werden vom Interpreter gespeichert: **(globale) Umgebung**.

Umbenennungen

Umbenennungen (nochmalige Benennungen mit dem gleichen Namen) sind nicht erlaubt:

```
> (define radius 11)

define: cannot redefine name: radius
```

Wir können also auf diese Weise nicht einfach den Umfang oder die Fläche für Kreise mit **anderen** Radien berechnen.

Wohl können wir aber bei der ursprünglichen Definition den Wert ändern. Soll zum Beispiel die genauere Approximation **`3.14159265358979323 85`** für π verwendet werden, so muss man dies nur an **einer** Stelle ändern!

Auswertung von Kombinationen (1)

Die verschiedenen **Ausdrücke** bilden zusammen mit ihren **Auswertungs-regeln** die **Syntax** einer Programmiersprache.

Auswertungsregel einer Kombination

Schritt 1: Werte die **Teilausdrücke** der Kombination aus.

Schritt 2: Wende die **Prozedur**, die sich als Wert des ersten (links stehenden) Teilausdruckes (Operation) ergibt, auf die Argumente an, die sich als Werte der anderen Teilausdrücke (Operanden) ergeben.

Ein Teilausdruck einer Kombination kann eine Kombination sein. Die Auswertungsregel ist somit **rekursiv** (die Definition bezieht sich auf sich selbst).

Auswertung von Kombinationen (2)

Bzgl. Schritt 2 der Auswertungsregel gilt:

- Der Wert eines **Zahlzeichens** ist die benannte Zahl.
- Der Wert eines eingebauten **Operators** ist die Folge von Maschinenbefehlen, welche die entsprechende Operation ausführen.
- Der Wert anderer **Namen** ist das Objekt, das in der Umgebung mit dem Namen verknüpft ist.

Auswertungsprozess: Baumakkumulation

Beispiel

```
(* (+ 2 (* 4 6))
(+ 3 5 7))
```

Auswertung von Kombinationen (3)

- Die Umgebung ist für die Bedeutung von Namen (Symbolen) in einem Ausdruck entscheidend. Sie definiert den Kontext, in dem Auswertungen statt finden.
- Definitionen mit **`define`**, z.B. (**`define x 3`**), sind keine Kombinationen. Solche **Sonderformen** haben ihre eigene Auswertungsregel.

Zusammengesetzte Prozeduren

Abstraktion durch die Definition von Prozeduren, d.h. die Benennung zusammengesetzter Operationen.

Beispiel:

Definition: Quadrieren einer Zahl: Multipliziere die Zahl mit sich selbst.

In *Scheme*:

```
(define (quadrat x) (* x x))
```

Diese Definition erzeugt eine Prozedur zur Berechnung des Quadrats einer Zahl **x** und verknüpft diese Prozedur mit dem Namen **`quadrat`**.

x ist ein lokaler Name, der das Argument (Operand) der Prozedur (Operation) in der Definition der Prozedur bezeichnet (Platzhalter für einen unbekannten Wert).

Anwendung der Prozedur `quadrat`

```
> (quadrat -3.2)

10.24
```

Offenbar ist die Operation, die der Name **`quadrat`** benennt, mit der Zahl **`-3.2`** an der Stelle von **`x`** ausgeführt worden.

Prozedurdefinition (1)

Allgemeine Form einer Prozedurdefinition:

```
(define (<name> <formale parameter>) <rumpf>)
```

Es gilt:

- Der **<*name*>** ist mit der Prozedurdefinition in der Umgebung verknüpft.
- Die **<*formalen parameter*>** benennen die Argumente der Prozedur innerhalb des Prozedurrumpfes.
- Der **<*rumpf*>** ist ein Ausdruck, der den Wert einer Anwendung der Prozedur auf **aktuelle Parameter** liefert, sobald diese für die formalen Parameter eingesetzt worden sind.

Prozedurdefinition (2)

Gemäß der Prozedurdefinition kann **`quadrat`** nicht auf zwei (oder mehr) Argumente angewendet werden (**Syntaxfehler**):

```
> (quadrat 3 2)
procedure quadrat: expects 1 argument, given 2: 3 2
```

Der aktuelle Parameter kann ein beliebiger Ausdruck sein, der eine Zahl als Wert hat:

```
> (quadrat (+ 3 2))
25
> (quadrat (quadrat 2))
16
```

Verwendung von quadrat als „Baustein“ (1)

- Verwendung von quadrat als “Baustein” in einer (beliebigen) Kombination:

```
> (+ (quadrat (quadrat 2)) (quadrat 4))
32
```

- Verwendung von quadrat als “Baustein” in einer anderen Prozedur:

```
(define (quadratsumme x y)
        (+ (quadrat x) (quadrat y)))
> (quadratsumme 3 4)
25
```

Verwendung von `quadrat` als „Baustein“ (2)

- Weiterverwendung dieser “Bausteine”:

```
(define (quadratsumme-hoch-2 x y)
     (quadrat (quadratsumme x y)))
> (quadratsumme-hoch-2 3 4)
625
```

Zusammengesetzte Prozeduren werden genau so wie elementare Prozeduren verwendet!

Substitutionsmodell für Prozeduranwendungen

Modell (Denkmuster, Schema) zur Ermittlung der Bedeutung einer Prozeduranwendung.

Auswertung von Kombinationen mit zusammengesetzten Prozeduren als Operatoren:

- Werte die Elemente der Kombination aus und wende die Prozedur (Wert des Operators der Kombination) auf die Argumente (Werte der Operanden der Kombination) an.

Anwendung einer zusammengesetzten Prozedur auf ihre Argumente:

- Werte den Rumpf der Prozedur aus, nachdem jeder formale Parameter durch das entsprechende Argument ersetzt wurde.

Der Mechanismus zur Anwendung elementarer Prozeduren auf ihre Argumente ist schon vorhanden.

Auswertungsmodelle

- Auswertung in **applikativer** Reihenfolge: Werte die Argumente aus und wende dann die Prozedur an (Modell gemäß Beispiel oben).
- Auswertung in **normaler** Reihenfolge:

```
(f 5)
(quadratsumme (+ 5 1) (* 5 2))
(+ (quadrat (+ 5 1)) (quadrat (* 5 2)) )
(+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)) )
(+ (* 6 6) (* 10 10) )
(+ 36 100)
136
```

→ Weniger effizientes Modell, aber als Technik in manchen Situationen nützlich.

→ Beide Auswertungsmodelle führen zu identischen Ergebnissen.

Beispiel: Schema für Programmentwurf (1)

Berechne die Fläche einer ringförmigen Kreisscheibe mit gegebenem inneren und äußeren Radius:

(1) Verstehen des Anwendungszwecks des Programms:

- Was sind die Eingangsdaten?
- Welche Daten werden produziert?
- Festlegung eines sinnvollen Programmnamens.
- Kurzform: ;; ring-flaeche: Zahl Zahl ---> Zahl
  → Zeilen, die mit **;;** beginnen, sind Kommentare.

(2) Festlegung des Prozedurkopfes. Sinnvolle Benennung der formalen Programmparameter gemäß den Eingangsdaten.

- Prozedurkopf (header)

```
(define (ring-flaeche aussen innen) …)
```

Beispiel: Schema für Programmentwurf (2)

(3) Hinzufügen einer Kurzbeschreibung dessen, was das Programm berechnet (Anwendungshintergrund, –zweck)

- ;; berechnet die Fläche einer ringförmigen Kreisscheibe mit den Radien innen und außen

(4) Abgabe von Beispielen für die Anwendung des Programms

- ;; Beispiel: (ring-flaeche 5 3) ergibt 50.24
- Es ist sinnvoll, dies vor dem Schreiben des eigentlichen Programms (nächster Punkt) zu tun!

(5) Ausfüllen des Rumpfes zur Berechnung (gemäß den Beispielen)

- Ausgangsdaten aus den Eingangsdaten mit Operationen von *Scheme* oder Hilfsfunktionen (schon geschrieben oder noch zu schreiben).

Beispiel: Schema für Programmentwurf (3)

```
;; Definition:
(define (ring-flaeche aussen innen)
    (- (kreis-flaeche aussen)
       (kreis-flaeche innen)))
;; Hilfsvariable
(define Pi 3.14159)

;; kreis-flaeche: Zahl --> Zahl
;; berechnet die Flaeche eines Kreises mit Radius radius
(define (kreis-flaeche radius)
    (* Pi (* radius radius)))
```

Beispiel: Schema für Programmentwurf (4)

(6) Angabe von Tests zur Detektion von **Laufzeitfehlern** (z. B. Division durch Null während der Auswertung eines Ausdruckes) und **logischen Fehlern** (das Programm tut nicht was es soll).

Beispiel: Zusammensetzen von Funktionen (1)

Das Problem

Ein Kinobesucher hat festgestellt, dass bei einem Eintrittspreis von 5,- Euro (im Mittel) 120 Besucher eine Vorstellung besuchen. Verringert er den Preis um 10 Cent, so kommen (im Mittel) 15 Besucher mehr in eine Veranstaltung. Eine Vorstellung kostet den Besitzer 180,- Euro, und jeder Besucher verursacht 4 Cent Unkosten.

Der Besitzer hätte gerne ein Programm, das ihm erlaubt bei Eingabe eines Eintrittspreises den Gewinn (im Mittel) pro Vorstellung zu berechnen.

Beispiel: Zusammensetzen von Funktionen (2)

Analyse der Problemstellung

(1) Der Gewinn ist gleich dem Umsatz abzüglich der Kosten.

(2) Der Umsatz entspricht dem Eintrittspreis mal der Anzahl der Besucher.

(3) Die Kosten sind 180,- Euro plus ein Betrag, der von der Anzahl der Besucher abhängt.

(4) Die Anzahl der Besucher hängt vom Eintrittspreis ab.

Beispiel: Zusammensetzen von Funktionen (3)

Definieren dieser funktionalen Zusammenhänge:

;; dieses Programm ist leserlich

```
(define (gewinn eintrittspreis)
   (- (umsatz eintrittspreis)
      (kosten eintrittspreis)))
(define (umsatz eintrittspreis)
   (* (besucherzahl eintrittspreis) eintrittspreis))
(define (kosten eintrittspreis)
   (+ 180 (* 0.04 (besucherzahl eintrittspreis))))
(define (besucherzahl eintrittspreis)
   (+ 120 (* (/ (- 5 eintrittspreis) 0.1) 15)))
```

Beispiel: Zusammensetzen von Funktionen (4)

Funktionale Zusammenhänge ineinander substituieren:

```
;; dieses Programm ist nicht leserlich
(define (gewinn2 eintrittspreis)
   (+ -180
    (* -0.04
     (+ 120
      (* 150
       (+ 5
        (* -1 eintrittspreis)))))
    (* (+ 120
       (* 150
        (+ 5
         (* -1 eintrittspreis))))
       eintrittspreis)))
```

Beispiel: Zusammensetzen von Funktionen (5)

Ergebnis:

```
> (gewinn 1)
511.2
> (gewinn 2)
937.2
> (gewinn 3)
1063.2
> (gewinn 4)
889.2
> (gewinn 5)
415.2
```

Offenbar liegt der optimale Eintrittspreis zwischen 2,- und 4,- Euro. (Um diesen genau zu bestimmen, hilft nur Mathematik: 2.92 Euro).

Bedingte Ausdrücke und Prädikate

Oft hängt die Verarbeitung von Informationen von Bedingungen ab. Zur Darstellung, ob Bedingungen erfüllt sind oder nicht, benötigen wir (neben den bisherigen Zahlen):

- Bool'sche Wahrheitswerte: **`true`**, **`false`**
- Prädikat: Prozedur oder Ausdruck, deren Auswertung den Wert **`true`** oder **`false`** liefert.
- Relationale Operationen zur Darstellung elementarer Prädikate mit Zahlen: **`<`**, **`>`**, **`=`**, **`<=`**, **`>=`**

Beispiel:

```
> (< 5 3)
false
> (= (- 5 3) (+ 1 1))
true
```

Logische Konnektoren

Logische Verknüpfungsoperatoren (**Konnektoren**) **and**, **`or`** (Sonderformen) und **`not`** zur Bildung zusammengesetzter Prädikate:

- (**and** $\langle p_1 \rangle ... \langle p_n \rangle$)

  Auswertung der Prädikate $\langle p_i \rangle$ von links nach rechts; sobald ein Prädikat den Wert **`falsch`** liefert, bricht die Auswertung ab und liefert den Wert **`falsch`**; liefert jedes Prädikat den Wert **`wahr`**, dann ist der resultierende Wert **`wahr`**.

- (**`or`** $\langle p_1 \rangle ... \langle p_n \rangle$)

  Auswertung der Prädikate $\langle p_i \rangle$ von links nach rechts; sobald ein Prädikat den Wert **`wahr`** liefert, bricht die Auswertung ab und liefert den Wert **`wahr`**; liefert jedes Prädikat den Wert **`falsch`** , dann ist der resultierende Wert **`falsch`** .

- (not $\langle a \rangle$)

  Der resultierende Wert ist **`wahr`** wenn $\langle a \rangle$ den Wert **`falsch`** liefert, bzw. umgekehrt.

Logische Konnektoren (Bsp.)

```
> (and (< 3 5) (>= 4 4))
true
> (or false (and (< 3 5) (>= 4 4)))
true

(define (im-interval? x unten oben)
(and (>= x unten) (<= x oben)))

> (im-interval? 2.1 2 3)
true
> (im-interval? 3 2 3)
true
> (im-interval? -2.5 2 3)
false
```

Bedingte Ausdrücke (1)

Einführendes Beispiel

Definition des Absolutbetrages einer Zahl x durch Fallunterscheidung

$$|x| = \begin{cases} x, & x > 0 \\ 0, & x = 0 \\ -x, & x < 0 \end{cases}$$

Sonderform `cond`:

```
(define (betrag x)
        (cond   ((> x 0) x)
                ((= x 0) 0)
                ((< x 0) (- x)) ) )
```

Bedingte Ausdrücke (2)

Allgemeine Form des bedingten Ausdruckes:

(cond $(\langle p_1 \rangle \langle a_1 \rangle)$
$(\langle p_2 \rangle \langle a_2 \rangle)$

…

$(\langle p_n \rangle \langle a_n \rangle)$)

$(\langle p \rangle \langle a \rangle)$	Klausel
$\langle p \rangle$	Bedingung, Test oder Prädikat
$\langle a \rangle$	Folgeausdruck

Auswertung bedingter Ausdrücke

- Werte nacheinander die Bedingungen oder Test $\langle p_1 \rangle$, $\langle p_2 \rangle$, … aus, bis ein Prädikat $\langle p_i \rangle$ den Wert **`wahr`** ergibt. Liefere den Wert des entsprechenden Folgeausdrucks $\langle a_i \rangle$ der Klausel als Wert des bedingten Ausdrucks. Der Wert von **`cond`** ist nicht definiert, wenn keines der Prädikate wahr ist.
- **`else`**-Klausel: (**`else`** $\langle a \rangle$)

  Optionale letzte Klausel eines bedingten Ausdrucks; $\langle a_i \rangle$ wird ausgewertet, wenn alle Prädikate den Wert **`falsch`** liefern.

Beispiel

```
(define (betrag x)
        (cond ((< x 0) (- x))
        (else x) ) )
```

Sonderform if (1)

`(if <Prädikat> <Folge> <Alternative> )`

Zur Unterscheidung von genau zwei Fällen.

```
(define (betrag x)
   (if (< x 0) (- x) x))
```

Beispiel

;; Language: Standard (R5RS)

;; n: ganze Zahl > 0

```
(define (countdown n)
   (display n)
   (newline)
```

(Fortsetzung)

Sonderform `if` (2)

```
   (if (<= n 0)
       (display "los!\n")
       (countdown (- n 1)))) ;; rekursiver Aufruf!
> (countdown 3)
3
2
1
0
los!
```

Mit **`countdown`** haben wir ein erstes Beispiel für die rekursive Definition einer Prozedur: **`countdown`** ruft sich selbst auf!

Der wiederholte Aufruf der Prozedur führt zu einem iterativen Prozess, bei dem ein Zustand – hier beschrieben durch die Variable **n** – verändert wird, bis ein Abbruchkriterium erfüllt ist.

Beispiel (1)

Iterative Berechnung der Quadratwurzel $y = \sqrt{x}$ (Newton-Verfahren)

Gegeben sei eine Zahl $x \geq 0$. Berechne eine Zahl $y \geq 0$, so dass

$$g(y) := y^2 - x = 0 \qquad (1.1)$$

Sei bspw. $x = 2$. Dann liegt y zwischen 1 und 2, da $1^2 - 2 \leq 0 \leq 2^2 - 2$ ist. Wir können y also ausdrücken als

$$y = y_0 \text{ (Schätzwert)} + dy_0 \text{ (Korrekturterm)} \qquad (1.2)$$

mit bspw. $y_0 = 1$. Einsetzen in Gleichung (1.1) und Abschätzen des Korrekturterms:

$$g(y) = g(y_0 + dy_0) = 0 \approx g(y_0) + g'(y_0)dy_0$$

$$\Rightarrow \quad dy_0 \approx -\frac{g(y_0)}{g'(y_0)} = -\frac{y_0^2 - x}{2y_0} = -\frac{1}{2}y_0 + \frac{1}{2}\frac{x}{y_0}$$

Beispiel (2)

Damit kann gemäß Gleichung (1.2) ein verbesserter Schätzwert berechnet werden:

$$y_1 = y_0 + dy_0 = \frac{1}{2}\left(y_0 + \frac{x}{y_0}\right)$$

Analog verbessert man sukzessive $y_1, y_2, \ldots$ bis $\left|g(y_k)\right| = \left|y_k^2 - x\right| < \varepsilon$

```
;; berechnet iterativ die Quadratwurzel von x nach dem Newton-Verfahren
(define (wurzel x)
   (wurzel-iter 1.0 x))
```

Beispiel (3)

```
;; verbessert den Schätzwert einer Quadratwurzel bis er gut genug ist
(define (wurzel-iter schaetzwert x)
   (if (wurzel-gut-genug? schaetzwert x)
      schaetzwert
      (wurzel-iter (wurzel-verbessern schaetzwert x) x)))

;; berechnet einen besseren Wert des Schätzwerts einer Quadratwurzel
(define (wurzel-verbessern schaetzwert x)
   (+ schaetzwert (wurzel-korrekturterm schaetzwert x)))
```

Beispiel (4)

```
;; berechnet den Korrekturterm zum Verbessern einer Quadratwurzel
(define (wurzel-korrekturterm schaetzwert x)
(/ (+ (- schaetzwert) (/ x schaetzwert)) 2))

;; prüft, ob das Residuum der Wurzelgleichung unter einer Schwelle liegt
(define (wurzel-gut-genug? schaetzwert x)
   (< (abs (- (quadrat schaetzwert) x)) 0.001))
(define (quadrat x)
   (* x x))
```

Prozeduren als “black box”–Abstraktionen (1)

Der Rumpf der Prozedur **`quadrat`** ist nicht eindeutig. Eine andere Möglichkeit wäre:

```
(define (quadrat x)
   (expt x 2))
```

Wir beobachten:

- Für die Definition der Prozedur **`wurzel-gut-genug?`** ist nur die Existenz einer Methode zur Berechnung von x^2 entscheidend, nicht ihre Implementierung (**prozedurale Abstraktion**).
- Details der Implementierung möchte der Benutzer in der Regel nicht wissen.
- Insbesondere sind die **lokalen** Namen, die bei der Implementierung einer Prozedur für die formalen Parameter benutzt werden, “außerhalb” von **`quadrat`** ohne Bedeutung. Ihr **Geltungsbereich** ist auf den Rumpf der Prozedur beschränkt. Die Bedeutung der Prozedur hängt nicht von dieser Benennung ab.

Prozeduren als “black box”–Abstraktionen (2)

Folgendes ist also in diesem Sinne gleichwertig:

```
(define (quadrat x) (* x x))

(define (quadrat y) (* y y))
```

Da die Definition der Prozedur an die Benennung der formalen Parameter gebunden ist, spricht man von **gebundenen** Variablen, im Gegensatz zu **freien** Variablen. In **`wurzel-gut-genug?`** sind **`schaetzwert`** und **`x`** gebunden, **`<`**, **`-`**, **`abs`** und **`quadrat`** hingegen frei.

Prozeduren als “black box”–Abstraktionen (3)

Die Bedeutung der Prozedur ändert sich nicht, wenn eine gebundene Variable **konsistent** (durchgängig) umbenannt wird, wohl aber bei der Umbenennung einer freien Variablen. So kann **`abs`** nicht einfach durch **`expt`** ersetzt werden.

Wäre der Geltungsbereich gebundener Variablen nicht auf den Prozedurrumpf beschränkt, dann wäre **`x`** in

```
(define (wurzel-gut-genug? schaetzwert x)
   (< (abs (- (quadrat schaetzwert) x)) 0.001))
```

nicht ein anderes **`x`** als in

```
(define (quadrat x)
   (* x x))
```

mit unerfreulichen Folgen: **`quadrat`** wäre keine “black box”!

Als Konsequenz davon benutzen wir **interne Definitionen** und verstecken alles in **`wurzel`**, was nur für die Implementierung relevant ist.

Beispiel

```
;; Language: Standard (R5RS)
(define (wurzel x)
   (define (wurzel-iter schaetzwert)
      (if (wurzel-gut-genug? schaetzwert)
      schaetzwert
      (wurzel-iter (wurzel-verbessern schaetzwert))))
(define (wurzel-verbessern schaetzwert)
   (+ schaetzwert (wurzel-korrekturterm schaetzwert)))
(define (wurzel-korrekturterm schaetzwert)
   (/ (+ (- schaetzwert) (/ x schaetzwert)) 2))
(define (wurzel-gut-genug? schaetzwert)
   (< (abs (- (quadrat schaetzwert) x)) 0.001))
(define (quadrat y)
   (* y y))
(wurzel-iter 1.0))
```

Lexikalische Bindungsregel

Darüber hinaus vereinfachen sich aufgrund der **Blockstruktur** die internen Definitionen, da **x** in dem Geltungsbereich von **`wurzel`** gebunden ist und nicht mehr jeder Prozedur übergeben werden muss. Vielmehr taucht **x** als freie Variable in den Rümpfen der internen Prozeduren auf und es gilt die lexikalische Bindungsregel.

Lexikalische Bindungsregel: Die freie Variable **x** erhält den Wert des entsprechenden Argumentes der übergeordneten Prozedur.

Vor- und Nachteile der funktionalen Programmierung

Vorteile

- Mathematische Korrektheitsbeweise sind möglich.
- Der Programmierer wird von komplizierter Syntax befreit, die nicht zur eigentlichen Beschreibung oder Lösung des Problems dient.

Nachteile

- Die Übersetzung erzeugt sehr große und komplexe Datenstrukturen, insbes. zum Festhalten des Programmzustands bei der Rekursion.
- Funktionale Programme laufen oft langsam.
- Viele Probleme, insbesondere aus kommerziellen Anwendungen, lassen sich nicht leicht funktional beschreiben.
- Für Ungeübte schwer erlernbar.