Programmierparadigmen

Software Entwicklung 1

Annette Bieniusa, Mathias Weber, Peter Zeller

1 Imperative vs. Deklarative Programmierung

Wir haben in SE1 bisher zwei Programmiersprachen näher betrachtet: Java und C. Wie bereits diese beiden Beispiele zeigen, bauen Programmiersprachen auf z.T. sehr unterschiedlichen Konzepten auf, wie Daten und Berechnungen repräsentiert und implementiert werden. Diese prägen die Semantik von Programmen und erfordern einen bestimmten Programmierstil.
Wir stellen im Folgenden die beiden wichtigsten Kategorien von Programmierparadigmen vor:

Imperative Programmierung In der imperativen Programmierung ist ein Programm eine Folge von Anweisungen / Befehlen, die in einer vorgegeben Reihenfolge abgearbeitet werden. Diese Art der Programmierung ist eng an die Arbeitsweise von Prozessoren angelehnt. Viele Programmiersprachen, z.B. Maschinensprache/Assembler, Java und C, sind im Kern imperativ. Der Programmablauf ist dabei oft durch Kontrollstrukturen (Verzweigungen, Schleifen, etc.) strukturiert. Als Abstraktionsmechanismus stehen häufig Prozeduren zur Verfügung.

Deklarative Programmierung Bei der imperativen Programmierung steht im Vordergrund, *wie* etwas berechnet wird. Die deklarative Programmierung konzentriert sich hingegen darauf, *was* berechnet wird. Dieses Paradigma beruht häufig auf mathematischen Theorien, die es erlauben Eigenschaften des Programms wie Korrektheit vergleichsweise einfach nachzuweisen.
Beispiele für deklarative Programmiersprachen:

- Die funktionale Programmiersprache Haskell basiert auf dem Lambda-Kalkül (siehe nächster Abschnitt). Hier eine deklarative Variante von Quicksort in Haskell:

```
quicksort [] = []
quicksort (x:xs) = quicksort [n | n <- xs, n < x] ++ [x] ++
    quicksort [n | n <- xs, n >= x]
```

- Die Datenbanksprache SQL (Structured Query Language) basiert auf der relationalen Algebra; sie definiert die Struktur von Datenbanktabellen, ermittelt und modifiziert Einträge. Ein Beispiel für die Erhöhung aller Einkommen der Angestellten einer Firma könnte wie folgt ausgeführt werden:

```
UPDATE salary
  SET income = income + 0.1 * income
  WHERE status = 'employed'
```

- Die logische Programmiersprache Prolog basiert auf Logik erster Stufe; sie verwendet Fakten und Regeln, um Anfragen an eine Datenbasis zu beantworten.

```
father(abraham, isaac).
mother(sarah,isaac).
male(isaac).
son(X,Y) :- father(Y,X), male(X).
```

  Die Anfrage, wer der Sohn von Isaac ist, wird wie folgt aufgelöst:

```
?- son(X,abraham).
X = isaac
```

Für die beiden Kategorien gibt es verschiedene Ausprägungen (objekt-orientiert, prozedural, aspekt-orientiert, etc.). Die meisten Programmiersprachen sind nicht streng an ein Paradigma gebunden, sondern prägen im Laufe ihrer Entwicklung oft Aspekte verschiedener Paradigmen aus.

2 Funktionale Programmierung

Funktionale Programmierung ist ein deklaratives Programmierparadigma, bei dem Programme im Kern aus Funktionen und Funktionsanwendungen bestehen. Der Funktionsbegriff hier orientiert sich an mathematischen Funktionen (nicht an Funktionen wie in C): Eine Funktion bildet Eingabewerte auf Ausgaben ab. Für die gleichen Eingaben wird bei jedem Funktionsaufruf die gleiche Ausgabe geliefert.
Die wichtigsten Eigenschaften der funktionalen Programmierung sind:

Funktionen als Werte Funktionen können an Variablen gebunden werden, als Parameter übergeben werden und auch als Rückgabe einer Funktion verwendet werden.

Referentielle Transparenz Ausdrücke haben einen eindeutigen Wert, der sich weder durch Auswertung noch anderweitig ändern kann und der ausschließlich von den Werten der Teilausdrücke abhängt. Die Auswertung von Ausdrücken liefert ausschließlich den Wert und hat keine weiteren Effekte.

Daraus ergeben sich folgende Eigenschaften für die Auswertung von Programmen:

Referentielle Transparenz erlaubt eine flexible Auswertungsstrategie für Ausdrücke. Ein Ausdruck kann durch einen Ausdruck gleichen Werts ersetzt werden, ohne dass dies Auswirkungen auf die Programmsemantik hat.

Die Reihenfolge der Auswertung von Teilausdrücken hat keinen Effekt auf den Wert des Gesamtausdrucks.

Bei referentieller Transparenz sind die linke und rechte Seite des folgenden Beispiels äquivalent:

```
int a = 2;
int x = f(a);
int y = x + x;
```

```
int a = 2;
int x = f(a);
int y = f(a) + f(a);
```

In rein-funktionalen Sprachen ist referentielle Transparenz immer der Fall.[1] Bei Java jedoch gilt die Umformung zum Beispiel bei folgender Definition von `f` nicht:

```
static int y = 0;
static int f(a) {
  y++;
  return a + y;
}
```

Frage 1: Welche Werte nehmen `x` und `y` für das linke und das rechte Beispiel mit der gegebenen Java-Implementierung an?

2.1 Der Lambda-Kalkül

Der Lambda-Kalkül ist eine formale Sprache mit Regeln zur Auswertung von Funktionen.

Syntax Die Syntax der Lambda-Terme kann dabei durch die folgende Grammatik definiert werden:

$$
\begin{aligned}
\Gamma &= (N, T, \Pi, T) \\
N &= \{E\}, \\
T &= \{\lambda, \rightarrow, (\ ,\ ), id\}, \\
\Pi &= \left\{ \begin{array}{lcl} E & \rightarrow & id \\ E & \rightarrow & (\lambda\ id \rightarrow E) \\ E & \rightarrow & (E\ E) \end{array} \right\}
\end{aligned}
$$

[1] Seiteneffekte sind allerdings notwendig für I/O-Operationen; funktionale Sprachen bestehen daher normalerweise aus einem rein-funktionalen Kern und zusätzlichen seiteneffektbehafteten Operationen (z.B. I/O-Monade in Haskell).

Dabei steht das Terminalsymbol *id* für ***Variablen/Bezeichner*** (hier bestehend aus Buchstaben und Zahlen). Terme der Form (λ *id* $\rightarrow$ *E*) sind ***Funktionsabstraktionen***, bei denen *id* den Parameternamen und *E* die Implementierung der Funktion angibt. Terme der Form (*E E*) stellen ***Funktionsaufrufe*** da, wobei der linke Term eine Funktion ist und der rechte Term der Parameter für den Funktionsaufruf.

Im Vergleich zu den Programmiersprachen C und Java entspricht eine Funktionsabstraktion einer Funktion mit einem Parameter, die jedoch keinen Namen hat (eine "anonyme Funktion"). Außerdem ist die Klammernsetzung anders: Ein Funktionsaufruf im Lambda-Kalkül hat die Syntax `(g x)`, in Java und C wird er als `g(x)` geschrieben.

Beispiel	**Bedeutung**
`(f x)`	Aufruf der Funktion `f` mit Parameter `x`
`(f (g x))`	Aufruf der Funktion `f` mit Parameter `(g x)`
`((f g) x)`	Aufruf der Funktion `(f g)` mit Parameter `x` (`(f g)` liefert eine Funktion)
`(λx → (f (f x)))`	Funktion, die ein `x` nimmt und zweimal die Funktion `f` auf `x` anwendet.
`((λx → (f (f x))) y)`	Aufruf der obigen Funktion mit Parameter `y`.

Um das Schreiben von größeren Termen zu vereinfachen, verwenden wir die folgenden Vereinbarungen:

1. Wenn die Implementierung einer Funktionsabstraktion wieder eine Funktionsabstraktion ist, können die Variablen zusammengefasst werden.
   Beispiel: `(λx y → t)` statt `(λx → (λy → t))`.

2. Wir betrachten die Funktionsanwendung als linksassoziativ, d.h. statt `((f x) y)` können wir `(f x y)` schreiben.

3. Die Klammern in Implementierungen von Funktionsabstraktionen können weggelassen werden. Statt `(λx → (f x))` schreiben wir kürzer `(λx → f x)`.

4. Äußere Klammern können weggelassen werden.
   Beispiel: `λx → f x` statt `(λx → f x)`.

Variablenbindung Die Verwendung einer Variable bezieht sich immer auf den gleichnamigen Parameter der nächsten umschließenden Funktionsabstraktion. Die Variable wird dann durch die Funktionsabstraktion ***gebunden***. Falls keine solche Parameterdeklaration existiert, nennen wir die Variable ***frei***.
Betrachten wir als Beispiel den Term `(λx → (λx → f x) x) x` .
Die erste Verwendung von `x` ist in der inneren Funktionsabstraktion gebunden, die zweite Verwendung von `x` ist an die äußere Abstraktion gebunden; und die letzte Verwendung von `x` sowie `f` sind freie Variablen.

Frage 2: Welche der Variablen sind im folgenden Term frei bzw. gebunden?

`(λf → (λx → f (x y))) ((λy → y) f)`

Substitution Die Schreibweise $s[t/x]$ (sprich: "Term s mit Term t für Variable x" oder "s mit t für x") bezeichnet den Term, der entsteht, wenn alle **freien** Variablen x in s durch den Term t ersetzt werden. Diese Ersetzung wird ***Substitution*** genannt. Eine Substitution ist **nicht erlaubt**, wenn eine freie Variable in `t` nach der Substitution gebunden wäre.
Beispiele:

- Die Substitution `((λx → f x) x)[(g y)/x]` ergibt `((λx → f x) (g y))`.
- Die Substitution `((λx → (λy → x y)) x)[y/x]` ist nicht erlaubt.

Semantik von Lambda-Ausdrücken Die Bedeutung (Semantik) von Lambda-Ausdrücken lässt sich mit Hilfe der Substitution und zwei Umformungsregeln erklären. Die Umformungsregeln sind dabei die α-Konversion und die β-Konversion.

Die ***α-Konversion*** erlaubt das Umbenennen von Parametern. Der Term `λx → t` lässt sich äquivalent umformen zum Term `λy → t[y/x]`, wenn y nicht bereits als freie Variable im Term `t` vorkommt.
Beispiel: `λa → f (f a)` ist äquivalent zu `λb → f (f b)`, aber nicht zu `λf → f (f f)`

Die ***β-Konversion*** erklärt die Auswertung von Funktionsaufrufen. Wenn der linke Term eines Funktionsaufrufs eine Abstraktion ist, so kann der Aufruf ausgewertet werden, indem die Implementierung verwendet wird und darin der Parameter durch den aktuell gegebenen Term ersetzt wird.
Formal lässt sich ein Ausdruck `(λx → s) t` äquivalent umformen zu `s[t/x]`, wenn diese Substitution erlaubt ist. Falls die Substitution nicht erlaubt ist, können zuvor noch α-Konversion zum Umbenennen von Parametern verwendet werden.
Beispiel: `(λx → plus x x) (f a)` wird zu `plus (f a) (f a)`

Ein Term ist in ***β-Normalform***, wenn keine β-Umformungen mehr möglich sind (auch nicht durch weitere Umbenennungen).
Beide Arten von Konversion (α und β) können jeweils auf beliebige Teilausdrücke angewandt werden. Die Reihenfolge der Auswertung hat dabei keinen Einfluss auf das Ergebnis. In manchen Fällen gibt es jedoch Reihenfolgen, welche zu unendlich langen Umformungen führen können, ohne dass eine β-Normalform erreicht wird. Es gibt auch Terme, die für keine Reihenfolge eine β-Normalform erreicht.

Auswertungsstrategien Es gibt zwei grundlegende Auswertungsstrategien, welche sich auch in verschiedenen Programmiersprachen wiederfinden.

call-by-value Es wird jeweils der innerste mögliche Aufruf ausgewertet.

call-by-name Es wird jeweils der äußerste / linkeste mögliche Aufruf ausgewertet.

Wie das folgende Beispiel zeigt, hat die Auswertung mit call-by-value den Vorteil, dass jeder Parameter nur einmal ausgewertet wird.

call-by-value	call-by-name
`(λx → plus x x) ((λx → mal x x) a)`	`(λx → plus x x) ((λx → mal x x) a)`
`(λx → plus x x) (mal a a)`	`plus ((λx → mal x x) a) ((λx → mal x x) a)`
`plus (mal a a) (mal a a))`	`plus (mal a a) ((λx → mal x x) a)`
	`plus (mal a a) (mal a a)`

Wegen diesem Vorteil wird die Auswertungsstrategie call-by-value von vielen Programmiersprachen, u.a. auch Java und C, verwendet.
Die Strategie call-by-name wird zum Beispiel von Haskell verwendet; sie wird auch "lazy" genannt, da sie den Vorteil hat, dass nur die Parameter ausgewertet werden, die auch verwendet werden.

call-by-value	call-by-name
`(λx → c) ((λx → e x x) a)`	`(λx → c) ((λx → e x x) a)`
`(λx → c) (e a a)`	`c`
`c`	

Mit der Strategie call-by-name lassen sich auch manche Programme auswerten, die mit call-by-value nicht terminieren würden.

Weitere Beispiele zum Lambda-Kalkül

Um interessantere Beispiele im Lambda-Kalkül zu rechnen, erweitern wir die Sprache um einige vordefinierte Funktionen. Diese Erweiterungen lassen sich alle auch direkt durch Funktionen darstellen[2].

- Ganze Zahlen $(0, -1, 1, -2, 2, \dots)$ und Boolsche Werte (`true`, `false`).
- Die aus Java und C bekannten mathematischen Operatoren. Der Ausdruck `x + y` entspricht im Lambda-Kalkül einem Funktionsaufruf `plus x y`.
- `if`-Ausdrücke der Form `if c then t1 else t2`. Dabei wird `t1` ausgewertet, wenn `c` sich zu `true` ergibt und andernfalls `t2`.

[2]Siehe zum Beispiel `https://en.wikipedia.org/wiki/Lambda_calculus#Encoding_datatypes`

Beispiel 1: Das folgende Beispiel zeigt die Auswertung einer Funktion mit zwei Parametern:

```
  (λx y → x + y) 3 4
= (λy → 3 + y) 4
= 3 + 4
= 7
```

Beispiel 2: Dieses Beispiel benötigt eine α-Konversion, bevor die β-Konversion angewandt werden kann:

```
  (λx y → (x + y)) y z
= (λx v → (x + v)) y z
= (λv → (y + v)) z
= y + z
```

Beispiel 3: Das folgende Beispiel zeigt, wie eine Funktion zum Verdoppeln von `y` zweimal auf die Zahl `3` angewendet wird:

```
  (λf x → f (f x)) (λy → y*2) 3
= (λx → (λy → y*2) ((λy → y*2) x)) 3
= (λy → y*2) ((λy → y*2) 3)
= (λy → y*2) (3*2)
= (λy → y*2) 6
= 6*2
= 12
```

Die Funktion `λf x → f (f x)` ist dabei eine Funktion, welche eine Funktion als Parameter nimmt. Funktionen dieser Art werden wir in Abschnitt 2.2 genauer betrachten.

Frage 3: Wozu wertet der folgende Term aus?

```
(λp → (λq → p q p)) (λx → (λy → x)) (λx → (λy → x))
```

Wertvereinbarungen Bisher haben wir Lambda-Ausdrücke kennen gelernt, die als Ergebnis einen bestimmten Wert berechnen. Dabei können die Ausdrücke recht komplex werden. Wir wollen diese komplexen Ausdrücke in einfache Teilausdrücke zerlegen. Diese Art der Abstraktion haben wir bereits als Motivation für Prozeduren kennengelernt: Prozeduren abstrahieren von bestimmten Ausführungsschritten. Beim Lambda-Kalkül abstrahieren wir dagegen von Teilausdrücken. Dazu will man in der Regel Funktionen definieren und dann zur Definition von komplexeren Funktionen verwenden. Als Beispiel betrachten wir hier eine Funktion, welche das Minimum von drei Zahlen berechnet.

```
(λx y z →
  if x <= y then
    if x <= z then x else z
```

```
  else
    if y <= z then y else z)
```

Diese Funktion lässt sich eleganter formulieren, indem wir von der Berechnung des Minimums zweier Zahlen abstrahieren. Wenn wir annehmen, dass wir eine Funktion `min` haben, die das Minimum zweier Zahlen berechnet, können wir die Funktion zur Berechnung des Minimums dreier Zahlen wie folgt schreiben:

```
(λx y z → min x (min y z))
```

Jedoch ist in obigem Lambda-Ausdruck der Bezeichner `min` nicht gebunden. Wir müssen also dem Bezeichner einen Wert zuordnen:

```
(λmin → (λx y z → min x (min y z)))
```

Wir müssen nun noch die Implementierung von `min` angeben, indem wir die entsprechende Funktion als Parameter übergeben und erhalten dabei folgenden Ausdruck:

```
(λmin → (λx y z → min x (min y z))) (λx y → if x <= y then x else y)
```

Da die Abstraktion von Teilausdrücken in Funktionen häufig vorkommt, wollen wir eine elegantere Syntax definieren. Eine ***Wertvereinbarung*** weist einem Bezeichner einen Wert zu: `let x = s in t` steht für den Term `(λx → t) s`.
Beispielsweise lässt sich die Funktion `min3` dann wie definieren und auf die Zahlen 6, 3 und 12 anwenden:

```
let min = (λx y → if x <= y then x else y) in
  let min3 = (λx y z → min x (min y z)) in
    min3 6 3 12
```

Um die Lesbarkeit zu erhöhen, können wir `let`-Terme um Parameter erweitern. Der Term `let f x = s in t` steht für `let f = (λx → s) in t`. Mit dieser Vereinbarung lässt sich das Beispiel wie folgt schreiben:

```
let min x y = if x <= y then x else y in
  let min3 x y z = min x (min y z) in
    min3 6 3 12
```

Rekursive Funktionen Das folgende Beispiel zeigt, dass der Lambda-Kalkül rekursive Funktionen berechnen kann. Wir definieren dazu abkürzend den Term `fix` als:

```
fix = λf → (λx → f (x x)) (λx → f (x x))
```

Für eine Funktion `f` gilt nun:

```
  fix f
= (λf → (λx → f (x x)) (λx → f (x x))) f
= (λx → f (x x)) (λx → f (x x))
= f ((λx → f (x x)) (λx → f (x x)))
= f (fix f)
```

Wir benutzen nun diese Gleichung, um die folgende Definition der Fakultäts-Funktion `fac` im Lambda-Kalkül auszuwerten.

```
facR = (λf x → if x == 0 then 1 else x * f (x - 1))
fac = fix facR
```

Wir können nun beispielsweise `fac 5` wie folgt auswerten:

```
  fac 5
= (fix facR) 5
= (facR (fix facR)) 5
= (facR fac) 5
= ((λf x → if x == 0 then 1 else x * f (x - 1)) fac) 5
= (λx → if x == 0 then 1 else x * fac (x - 1)) 5
= (if 5 == 0 then 1 else 5 * fac (5 - 1))
= (if false then 1 else 5 * fac (5 - 1))
= 5 * fac (5 - 1)
= 5 * fac 4
= ...
= 5 * 4 * fac 3
= 5 * 4 * 3 * fac 2
= 5 * 4 * 3 * 2 * fac 1
= 5 * 4 * 3 * 2 * 1 * fac 0
= 5 * 4 * 3 * 2 * 1 * 1
= 120
```

Wie dieses Beispiel zeigt, ist der Lambda-Kalkül mächtig genug um rekursive Funktionen zu definieren. Tatsächlich ist er mächtig genug, um alle berechenbaren Funktionen zu kodieren. Der Lambda-Kalkül ist die theoretische Grundlage der funktionalen Programmiersprachen. Er ist das funktionale Pendant zur Turing-Maschine, der theoretischen Grundlage für die imperative Programmierung.
Um das Definieren von rekursiven Funktionen zu vereinfachen erlauben wir, dass in einem `let`-Term `let x = s in t`, die Variable `x` im Term `s` verwendet werden darf. Solche Definitionen lassen sich mit der Funktion `fix` entsprechend umschreiben, wie wir oben gesehen haben. Mit dieser Vereinbarung lässt sich die Fakultätsfunktion wie folgt definieren und mit dem Parameter 5 aufrufen:

```
let fac x = if x == 0 then 1 else x * fac (x - 1) in fac 5
```

2.2 Funktionen höherer Ordnung

In der Informatik ist eine Funktion höherer Ordnung[3] eine Funktion, die eine (oder mehrere) Funktionen als Argument nimmt oder eine Funktion als Ergebnis liefert. Diese Art der Abstraktion ermöglicht die Parametrisierung einer Berechnung über eine andere Berechnung. Den Vorteil haben wir bereits bei der Einführung der Wertvereinbarung gesehen.
Für Funktionen höherer Ordnung gibt es in der Programmierung vielfältige Verwendung. Wir zeigen hier typische Anwendungen bei der Verarbeitung von Datensammlungen. Die `map`-Funktion wendet eine Funktion auf alle Elemente einer Datensammlung

[3] In Mathematik auch Funktional oder Operator genannt.

(z.B. Liste, Array, etc.) an. Hier ein Beispiel für die Verwendung von `map` auf einer Liste von Integern:

```
map (λx → x + 2) [1,2,3,4]   = [3,4,5,6]
map (λy → 3 * y) [1,2,3,4]   = [3,6,9,12]
```

Auch das Filtern von Listenelementen kann man einfach über eine Filterfunktion parametrisieren:

```
filter (λx → x < 10) [1,32,7]        = [1,7]
filter (λx → x % 2 == 0) [1,32,8]    = [32,8]
```

Eine weitere wichtige Funktion ist die Faltung von Listen; dabei werden die Listenelemente, ausgehend von einem Initialwert, mittels einer Funktion zu einem Wert "zusammengefaltet".

```
foldl (λy x → y + x) 0 [1,2,3]   = (((0 + 1) + 2) + 3) = 6
```

Wie die Funktionen `map`, `foldl` und `filter` und Listen im Lambda-Kalkül implementiert werden können, können Sie bei Interesse im Anhang Anhang nachlesen.

Frage 4: Was ist das Ergebnis der folgenden Aufrufe?

1. `map cook [` 🐮, 🍗, 🐔, 🌽 `]`
2. `filter isVegetarian [` 🍔, 🍟, 🍗, 🍿 `]`
3. `foldl feed` 😋 `[` 🍔, 🍟, 🍗, 🍿 `]`

Funktionspointer in C C erlaubt es Pointer auf Funktionen als Werte zu behandeln; dies emuliert das Verhalten von Funktionen höherer Ordnung. So nimmt z.B. die Sortierfunktionen `mergesort` und `qsort` der Standardbibliothek als Parameter eine Funktion, die zwei Elemente des zu sortierenden Arrays miteinander vergleicht.
Wir zeigen hier eine Implementierung einer Map-Funktion, die eine Funktion auf alle Einträge eines int-Arrays anwendet. Die `map_array`-Funktion nimmt als ersten Parameter einen Funktionspointer `f` auf eine Funktion, die einen Integer als Parameter erwartet; Rückgabetyp der Funktion ist `int` (vgl. Typdefinition von `intfunction`).
Die Funktion `f` wird nun auf alle Elemente des Arrays `ar` angewendet.

```
7  typedef int intfunction(int);
8
9  void map_array(intfunction f, int *ar, int size)
10 {
11   for (int i = 0; i < size; i ++)
12   {
13     ar[i] = f(ar[i]);
14   }
15 }
```

Die Abstraktion über die `map_array`-Funktion erlaubt es nun, Funktionen wie `doubling`, `increment` oder auch `print` auf die Einträge eines Arrays anzuwenden. Dabei spart man sich, für jede dieser Funktionen die Schleife über die Array-Elemente auszuprogrammieren.

```
19 int doubling(int i)
20 {
21   return i * 2;
22 }
23
24 int increment(int i)
25 {
26   return i + 1;
27 }
28
29 int print(int i)
30 {
31   printf("%i ", i);
32   return 0;
33 }
34
35 int main(void)
36 {
37     int n = 10;
38     int a[n];
39     // Initialisiere a
40     for(int i = 0; i < n; i++)
41     {
42         a[i] = i;
43     }
44
45     // Inkrementiere jeden Eintrag
46     map_array(increment, a, n);
47     // Verdopple jeden Eintrag
48     map_array(doubling, a, n);
49     // Gebe jeden Eintrag auf Konsole aus
50     map_array(print, a, n);
51     printf("λn");
52
53     return 0;
54 }
```

3 Zusammenfassung

Wir haben in diesem Kapitel einen kleinen Einblick in die Vielfalt der Programmiersprachen erhalten. Moderne Programmiersprachen bringen häufig mehrere Paradigmen zusammen (z.B. Scala als funktional-objektorientierte Sprache). In der Praxis

kommt der funktionalen Programmierung ein immer größerer Stellenwert zu, da die referentielle Transparenz die Programmierung von nebenläufigen Anwendungen für Multi-Core-Prozessoren stark vereinfacht.
Auch in Java finden sich seit Version 9 funktionale Programmierelemente in den Lambda-Ausdrücke und Methodenreferenzen. Diese behandeln wir im nächsten Kapitel.

Anhang: Zusatzmaterial zum Lambda Kalkül

Neben Zahlen und Boolschen Werten lassen sich auch Listen im Lambda-Kalkül darstellen. Wir verwenden hier die folgenden Funktionen:

- `nil` - Erstellt eine leere Liste
- `cons x l` - Erstellt eine Liste mit `x` als erstes Element und `l` als Rest der Liste.
- `head l` - Liefert das erste Element einer Liste `l`
- `tail l` - Liefert die Liste `l` ohne das erste Element
- `null l` - Ergibt `true`, wenn die Liste `l` leer ist, sonst `false`

Eine Liste lässt sich als eine Funktion darstellen, die zwei Parameter nimmt. Die leere Liste gibt immer den ersten Parameter zurück. Die nicht-leere Liste ruft immer den zweiten Parameter auf, wobei sie das erste Element der Liste und den Rest der Liste als Parameter übergibt. Mit dieser Kodierung lassen sich die oben genannten Funktionen wie folgt definieren:

```
nil = (λempty nonempty → empty)
cons x l = (λempty nonempty → nonempty x l)
head l = l undefined (λx xs → x)
tail l = l undefined (λx xs → xs)
null l = l true (λx xs → false)
```

Die folgende Funktion `concat` nimmt zwei Listen und hängt diese zusammen:

```
concat l1 l2 =
  if null l1 then
    l2
  else
    cons (head l1) (concat (tail l1) l2)
```

Die Funktion `listAdd1` nimmt eine Liste von Zahlen und erhöht jede Zahl in der Liste um 1:

```
listAdd1 l =
  if null l then nil
  else cons (1 + head l) (listAdd1 (tail l))
```

Die Funktion `listDouble` verdoppelt alle Zahlen in der Liste:

```
listDouble l =
  if null l then nil
  else cons (2 * head l) (listDouble (tail l))
```

Die Funktion map. Funktionen, die eine Operation auf alle Elemente in einer Liste anwenden, werden in der Praxis häufig benötigt. Da die Struktur solcher Funktionen sehr ähnlich ist, lässt sich diese in eine eigene Funktion abstrahieren, welche dann wiederverwendet werden kann. Dazu definieren wir eine Funktion höherer Ordnung `map`, welche eine Funktion `f` und eine Liste `l` nimmt und `f` auf alle Elemente in der Liste `l` anwendet:

```
map f l =
  if null l then nil
  else cons (f (head l)) (map f (tail l))
```

Mit Hilfe der Funktion `map` lassen sich dann `listAdd1` und `listDouble` wie folgt definieren:

```
listAdd1 l = map (λx → 1 + x) l
listDouble l = map (λx → 2 * x) l
```

Die Funktion filter. Eine oft verwendete Operation ist auch das Auswählen von Elementen aus einer Liste basierend auf einem gegebenen Filterkriterium. Dazu können wir die Funktion `filter` definieren, die eine Funktion `p` nimmt, welche das Filterkriterium darstellt.

```
filter p l =
  if null l then
    nil
  else
    if p (head l) then filter p (tail l)
    else cons (head l) (filter p (tail l))
```

Die Funktion foldl. Diese Funktion nimmt eine Funktion `f`, einen Startwert `s` und eine Liste `l`. Die Liste wird dabei von links nach rechts durchlaufen und der Startwert jeweils mit der Funktion `f` mit dem aktuellen Listenelement verrechnet.

```
foldl f s l =
  if null l then s
  else
    foldl f (f s (head l)) (tail l)
```