Kapitel 08: Rekursion und Terminierung

Software Entwicklung 1

Annette Bieniusa, Mathias Weber, Peter Zeller

Rekursion ist eine elegante Strategie zur Problemlösung, die es erlaubt eine Problemstellung auf eine gleichartige, aber kleinere Problemstellung zurückzuführen. Bei der rekursiven Formulierung eines Problems wird dabei die Definition wiederholt auf einfachere Instanzen des Problems angewandt, bis die Lösung des verbleibenden Problems so einfach ist, dass sie direkt angegeben werden kann. In der Informatik ist die Rekursion ein obligatorisches Mittel, um Algorithmen und Datenstrukturen zu definieren. In diesem Kapitel werden wir zunächst rekursive Prozeduren betrachten.
In Kapitel 04 haben wir gesehen, dass die Ausführung eines Programms ***terminiert***, wenn sie nach endlich vielen Schritten beendet ist. An Beispielen haben wir gesehen, dass "Endlosschleifen" die Terminierung eines Programms verhindern können. Auch Rekursion kann nicht-terminierenden Ausführungen verursachen. Wie kann man beweisen, dass eine Rekursion terminiert?

Lernziele dieses Kapitels:

- Rekursive Prozeduren zu charakterisieren.
- Terminierung von rekursiven Prozeduren mit Hilfe von geeigneten Abstiegsfunktionen zu beweisen.
- Lexikographische Ordnungen auf Zahlenpaaren anzuwenden.
- Iterative und rekursive Varianten eines Algorithmus zu vergleichen.

1 Rekursion

Nach der letzten WG-Party stapelt sich das dreckige Geschirr in der Küche[1]. Sie laufen unvorsichtigerweise an der Küche vorbei und jemand sagt Ihnen: "Erledigen Sie den Abwasch!" Was tun?
Um dem Geschirrberg Herr zu werden, entscheiden Sie sich für folgende Herangehensweise: Sie spülen ein einziges Teil und suchen dann die nächste Person, um ihr/ihm zu sagen, dass sie den Abwasch erledigen soll. Dieser Ansatz ist ganz angenehm, denn

[1] Quelle: `http://ais.informatik.uni-freiburg.de/teaching/ss09/info_MST/material/mst_08_recursion.pdf`

wenn alle diese Strategie wählen, muss keiner mehr als ein Teil spülen. Der Letzte muss gar nichts mehr machen - er findet ein leeres Spülbecken vor.
Diesen Algorithmus für die Aufgabe "Erledige den Abwasch"kann man folgendermaßen definieren:

- Falls das Spülbecken leer ist, ist nichts mehr zu tun.
- Andernfalls:
  - Spüle ein Teil;
  - Finde die nächste Person und sage ihr "Erledige den Abwasch".

Eine Prozedur dieser Art, die einen Teil der Aufgabe selbst löst und dann den Rest erledigt, indem sie sich selbst aufruft, wird ***rekursive Prozedur*** genannt. Dabei kann auch das Ergebnis des rekursiven Aufrufs zur Lösung des aktuellen Problems verwendet werden. Um zum Beispiel einen Stapel von Papieren zu sortieren, kann man folgendermaßen vorgehen:

- Falls der Stapel aus einem Blatt Papier besteht, so ist er schon sortiert.
- Andernfalls:
  - Teile den Stapel in zwei kleinere Stapel;
  - sortiere die kleineren Stapel;
  - füge die beiden sortierten Stapel wieder zu einem Stapel zusammen.

Wie die Beispiele zeigen, ist bei der Verwendung von Rekursion folgendes zu beachten:

- Die Problemgröße im aktuellen Aufruf kann durch einen diskreten Wert n beschrieben werden (z.B. Anzahl der Geschirrteile, Größe des Stapels).
- Die Größe des Teilproblems im rekursiven Aufruf muss kleiner sein als n.
- Die Zerlegung in Teilprobleme bricht irgendwann ab (z.B. wenn das Spülbecken leer ist, wenn der Stapel nur aus einem Blatt Papier besteht).

1.1 Terminologie

Unter Rekursion versteht man die Definition eines Problems, einer Struktur oder einer Funktion durch sich selbst. Um Rekursion genauer zu charakterisieren, führen wir hier zunächst einige Begriffe ein. Diese beziehen sich auf Prozeduren, sind aber analog auf Datenstrukturen, Methoden etc. anwendbar.

Definition Wir verwenden folgende Terminologie:

- Eine Prozedurdeklaration m heißt ***direkt rekursiv***, wenn der Prozedurrumpf einen Aufruf von m enthält.

- Eine Menge von Prozedurdeklarationen heißen ***verschränkt rekursiv*** oder ***indirekt rekursiv*** (engl. *mutually recursive*), wenn die Deklarationen gegenseitig voneinander abhängen.
- Eine Prozedurdeklaration heißt ***rekursiv***, wenn sie direkt rekursiv ist oder Element einer Menge verschränkt rekursiver Prozeduren ist.

Wie wir in späteren Kapiteln sehen werden, tritt Rekursion in der Informatik in natürlicher Weise bei Datenstrukturen wie Listen und Bäumen auf. Wir können auch aber auch rekursive Prozeduren verwenden, um Berechnungen auf Zahlen durchzuführen. Das klassische Beispiel für eine rekursive Prozedur ist die Berechnung der Fakultätsfunktion, die das Produkt der ersten n natürlichen Zahlen berechnet.

$$n! = \begin{cases} n * (n-1)! & \text{falls } n > 0 \\ 1 & \text{falls } n = 0 \end{cases}$$

```
public static int fac(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * fac (n-1);
  }
}
```

- Der ***Basisfall*** liefert einen Wert ohne einen nachfolgenden rekursiven Aufruf.
- Der ***Rekursionsschritt*** verwendet das Ergebnis von rekursiven Prozeduraufrufen für (andere) Parameterwerte für die Berechnung im aktuellen Prozeduraufruf.
- *Wichtig:* Der Basisfall muss nach endlich vielen rekursiven Aufrufen erreicht werden, damit das Programm terminiert.
  Dies ist in dieser Implementierung nur für Parameter $n \geq 0$ der Fall[2].

Lineare Rekursion Eine rekursive Prozedurdeklaration m heißt ***linear rekursiv***, wenn in jedem Ausführungszweig höchstens ein Aufruf von m auftritt. Die Prozedur zur Implementierung der Fakultätsfunktion ist ein typisches Beispiel für eine linear rekursive Prozedur.

Kaskadenartige Rekursion Eine rekursive Prozedurdeklaration m, die nicht linear rekursiv ist, heißt ***kaskadenartig rekursiv***. Das heißt, es muss in mindestens einem Ausführungszweig der Prozedur mehr als ein Aufruf von m auftreten.
Die Fibonacci-Folge ist eine Folge von natürlichen Zahlen, die bei diversen Naturphänomen in Erscheinung tritt (z.B. zur Modellierung von Hasenpopulationen). Jede Fibonacci-Zahl ist die Summe ihrer beiden Vorgänger:

[2] Bemerkung: Die Fakultätsfunktion wächst sehr rasch; bereits 13! lässt sich nicht mehr im Zahlenbereich von `int` darstellen. Wir lassen in diesem Fall Überläufe zur Vereinfachung außer Acht.

$$0 \quad 1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad 34 \quad 55 \quad 89 \quad 144 \quad 233 \ldots$$

$$fib(n) = \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ fib(n-1) + fib(n-2) & \text{für } n \geq 2 \end{cases}$$

```
public static int fib(int n) {
  if (n == 0) {
    return 0;
  }
  if (n == 1) {
    return 1;
  }
  return fib(n-1) + fib(n-2);
}
```

Repetitive Rekursion Eine linear rekursive Prozedurdeklaration m heißt ***repetitiv rekursiv*** (auch endrekursiv, engl. *tail recursive*), wenn jeder Aufruf von m in m der letzte auszuwertende Ausdruck ist.
Die Prozedur `ggT(m,n)` berechnet den größten gemeinsamen Teiler (ggT) von `m` und `n`. Sie ist repetitiv rekursiv, da in den beiden Rekursionsfällen der Aufruf von `ggT` der letzte auszuwertende Ausdruck ist.

```
public static int ggT(int m, int n) {
  if (m == n) {
    return m;
  } else if (m > n) {
    return ggT(m-n, n);
  } else {
    return ggT(m, n-m);
  }
}
```

Die Prozedur `fib()` hingegen ist nicht repetitiv rekursiv: Der letzte auszuwertende Ausdruck ist die Addition der Rückgabewerte der rekursiven Aufrufe.

> **Frage 1:** Auch die Prozedur zur Berechnung der Fakultätsfunktion ist in der obigen Implementierung nicht repetitiv-rekursiv. Wie kann man sie umschreiben, um eine repetitiv-rekursive Variante zu erhalten.

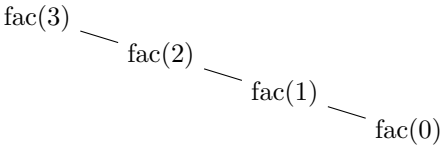
Die Ausführung von repetitiv-rekursive Prozeduren ist in einigen Programmiersprachen sehr effizient umgesetzt (z.B. Haskell) und dort von besonderer Bedeutung.

Verschränkte Rekursion Zuletzt nochmal ein Beispiel zur verschränkten Rekursion. Das folgende Prozedurpaar testet, ob eine (positive) Zahl gerade bzw. ungerade ist. Der rekursive Aufruf für `istGerade(5)` erfolgt dabei nicht in der Prozedur `istGerade` selbst, sondern indirekt beim Aufruf in der Prozedur `istUngerade`.

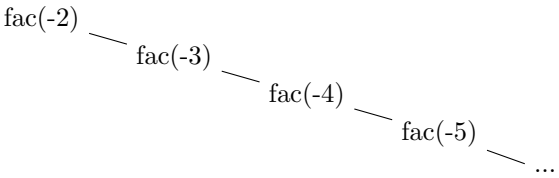
```
public static boolean istGerade(int n) {
  if (n == 0) {
    return true;
  } else {
    return istUngerade(n-1);
  }
}

public static boolean istUngerade(int n){
  if (n == 0) {
    return false;
  } else {
    return istGerade(n-1);
  }
}
```

2 Terminierung von rekursiven Funktionen

Angenommen, wir wollen die Fakultät von 3 berechnen. Dabei wird folgender Prozeduraufrufbaum erzeugt:



Was geschieht aber beim Aufruf von `fac(-2)`? Diese Ausführung liefert folgenden Prozeduraufrufbaum:



Die Ausführung für den Parameterwert -2 liefert eine unendliche Folge von rekursiven Aufrufen von `fac()`: Die Ausführung terminiert nicht[3], da der Basisfall mit Parameterwert 0 nicht erreicht wird. Es lässt sich dabei folgende Beobachtung machen: Der Abstand der Parameterwerte zum Basisfall wird beim Aufruf mit Parameterwert 3 mit jedem Aufruf kleiner, beim Aufruf mit Parameterwert -2 jedoch mit jedem Aufruf größer.
Die Idee den Abstand zum Basisfall zu "messen" kann man verwenden, um zu beweisen, dass eine rekursive Prozedur terminiert. Dazu kann man wie folgt vorgehen:
Sei `f` eine rekursive Prozedur, die durch eine Implementierung folgender Art gegeben ist:

[3] In Java benötigen Prozeduraufrufe Speicherplatz, so dass in der Praxis nach endlich vielen Aufrufen der Speicher voll ist und der Fehler `StackOverflowError` ausgelöst wird.

```
public static t f(t_1 x_1, ..., t_k x_k) {
  ... f(e_1, ..., e_k) ...
}
```

1. Man definiert mit einer Vorbedingung, welche Parameter erlaubt sind. Für diese gültigen Parameter soll die Prozedur terminieren.
2. Wir zeigen, dass der gültige Parameterbereich nicht verlassen wird. Das heißt, für jeden rekursiven Aufruf begründen wir, warum die neuen Parameter die Vorbedingung erfüllen.
3. Wir definieren eine ***Abstiegsfunktion*** h, welche den Abstand der Parameterwerte zu einem Basisfall misst. Dazu muss h jeder gültigen Kombination von Parameterwerten eine natürliche Zahl ($\mathbb{N}_0$) zuweisen. D.h. für die Prozedur `f` oben ist h eine Funktion $h : t_1 \times \cdots \times t_k \to \mathbb{N}_0$. Um sicherzustellen, dass das Ergebnis eine natürliche Zahl ist, kann der Betrag ($|x|$) oder das Maximum mit 0 ($max(0, x)$) verwendet werden. Für andere Ausdrücke sollte begründet werden, warum h eine natürliche Zahl liefert.
4. Wir begründen, warum der Wert von h bei jedem rekursiven Aufruf für die neuen Parameterwerte echt kleiner ist als für die Parameterwerte des aktuellen Aufrufs, also:
$$h(x_1, \ldots, x_k) > h(e_1, \ldots, e_k)$$

Da in $\mathbb{N}_0$ jede echt absteigende Kette endlich ist, ist eine unendliche Folge von rekursiven Aufrufen durch die Voraussetzungen ausgeschlossen. Wenn andere Ursachen zur Nichtterminierung (zum Beispiel Schleifen) ausgeschlossen werden können, dann ist damit die Terminierung der rekursiven Prozedur gezeigt.

Beispiel 1

```
1 public static int fac(int n) {
2   if (n == 0) {
3     return 1;
4   } else {
5     return n * fac (n-1);
6   }
7 }
```

1. Die Prozedur `fac` terminiert für positive Eingabewerte. Wie geben also die Vorbedingung `requires n >= 0` an.
2. Im rekursiven Aufruf in Zeile 5 wissen wir wegen der `if`-Anweisung, dass `n != 0` und daher gilt `n > 0`. Also ist `n-1 >= 0` und die Vorbedingung gilt.
3. Wir wählen $h(n) = n$ als Abstiegsfunktion. Da `n >= 0` für gültige Parameter gilt, bildet h die Parameter nach $\mathbb{N}_0$ ab.

4. Für den rekursiven Aufruf in Zeile 5 ist zu zeigen: $h(n) > h(n-1)$

   Dies gilt offensichtlich: $h(n) = n > n - 1 = h(n-1)$

Der Terminierungsbeweis für `fib` kann analog mit der gleichen Abstiegsfunktion geführt werden. Hierbei sind dann aber beide rekursiven Aufrufe zu betrachten:

$$\begin{aligned} h(n) &= n > n-1 = h(n-1) \\ h(n) &= n > n-2 = h(n-2) \end{aligned}$$

Beispiel 2

```
1  public static int ggT(int m, int n) {
2    if (m == n) {
3      return m;
4    } else if (m > n) {
5      return ggT(m-n, n);
6    } else {
7      return ggT(m, n-m);
8    }
9  }
```

1. Wir wählen als Vorbedingung `requires m > 0 && n > 0`.
2. Die Vorbedingung bleibt in den rekursiven Aufrufen erhalten:
   a) Beim Aufruf in Zeile 5 gilt `m > n`, also ist `m-n > 0`.
   b) Beim Aufruf in Zeile 7 gilt weder `m == n`, noch `m > n`. Also gilt `n > m` und damit ist `n-m > 0`.
3. Als Abstiegsfunktion wählen wir $h(m,n) = m + n$. Diese ist für $m, n > 0$ immer positiv.
4. Die Abstiegsfunktion wird für rekursive Aufrufe echt kleiner:
   a) Für den Aufruf in Zeile 5 gilt:
   $h(m,n) = m + n \overset{\text{weil } n>0}{>} m = m - n + n = h(m-n, n)$
   b) Für den Aufruf in Zeile 7 gilt:
   $h(m,n) = m + n \overset{\text{weil } m>0}{>} n = m + n - m = h(m, n-m)$

Beispiel 3

```
1  public static int f(int x, int y) {
2    if (x >= y) {
3      return x;
4    } else {
5      return f(2*x, y);
6    }
7  }
```

Die Prozedur `f(x, y)` berechnet die kleinste Zahl $z \geq y$, so dass $z = x \cdot 2^a$ für ein $a \in \mathbb{N}_0$.

1. Wir wollen zeigen, dass die Prozedur für Werte $x > 0$ terminiert, wählen also die Vorbedingung `requires x > 0`.
2. Wenn `x>0` gilt, dann auch `2*x>0`. Rekursive Aufrufe erfüllen also die Vorbedingung.
3. Die Idee ist, dass x in jedem rekursiven Aufruf größer wird und somit näher an den Basisfall $x >= y$ kommt. Dazu definieren wir die Abstiegsfunktion $h(x, y) = max(0, y - x)$. Wir verwenden hier das Maximum mit 0 um auszudrücken, dass der Basisfall für $x \geq y$ bereits erreicht ist.
4. Für den rekursiven Aufruf müssen wir jetzt zeigen, dass $h(2 \cdot x, y) < h(x, y)$ gilt, also dass die neuen Parameterwerte einen kleineren Wert der Abstiegsfunktion haben. Dies ist hier der Fall, da $2 \cdot x$ mit den gewählten Voraussetzungen immer größer als x ist[4].

Terminierung auf komplexeren Eingaben Das Nachweisverfahren zur Terminierung mit Abstiegsfunktionen ist nicht auf Prozeduren mit Zahlen beschränkt. Um die Terminierung von rekursiven Prozeduren auf Listen beispielsweise nachzuweisen, kann man die Länge der Liste zur Konstruktion von h zur Hilfe nehmen.

2.1 Verallgemeinerung der Abstiegsfunktion

Wir haben bis hier Abstiegsfunktionen verwendet, die Parameterwerte in die natürlichen Zahlen abbilden. Dieses Verfahren funktioniert korrekt, weil jede echt absteigende Kette in den natürlichen Zahlen endlich ist.[5]
Neben den natürlichen Zahlen mit der Ordnung $<$ gibt es noch andere (partiell) geordnete Mengen, in denen es keine unendlichen absteigenden Ketten gibt. Diese sogenannten noetherschen Ordnungen lassen sich daher ebenfalls für Terminierungsbeweise verwenden (siehe Zusatz-Informationen unten).
Das bisher verwendete Beweisschema passen wir nun so an, dass in Schritt 3 auch eine andere noethersche Ordnung gewählt werden kann. Diese muss dann aber explizit angegeben werden.

Lexikographische Ordnung Neben der Ordnung $(\mathbb{N}, \leq)$ sind auch Paare von natürlichen Zahlen mit der lexikografischen Ordnung noethersch $(\mathbb{N} \times \mathbb{N}, \leq_{lex})$. Die lexikografische Ordnung $\leq$ ist dabei wie folgt definiert:

$$(x_1, y_1) \leq_{lex} (x_2, y_2) \quad \text{genau dann, wenn (gdw.)} \quad x_1 < x_2 \quad \text{oder} \quad (x_1 = x_2 \text{ und } y_1 \leq y_2)$$

[4] Streng genommen sollten wir noch y einschränken, zum Beispiel $y < 2^{30}$, um Überläufe ausschließen zu können.

[5] Für $n_1 > n_2 > n_3 > \ldots$ wird irgendwann 0 erreicht und die Kette kann nicht unendlich lange fortgesetzt werden.

In der lexikographischen Ordnung wird also zuerst die erste Komponente verglichen und bei Gleichheit der ersten Komponente entscheidet die zweite Komponente über die Ordnung der Zahlenpaare. Zum Beispiel gilt $(1, 42) \leq_{lex} (2, 0)$ und $(3, 7) \leq_{lex} (3, 9)$, aber $(2, 1) \not\leq_{lex} (1, 100)$.

Beispiel: Ackermann-Funktion Die Ackermann-Funktion ist ein Beispiel für eine Funktion, bei der der zweite Parameterwert in rekursiven Aufrufen sehr schnell wächst. Sie wird in der theoretischen Informatik verwendet, um die Grenzen der Berechenbarkeit von Funktionen zu untersuchen.

```
1  static long ackermann(long m, long n) {
2    if (m == 0) {
3      return n + 1;
4    } else if (n == 0) {
5      return ackermann(m - 1, 1);
6    }
7    return ackermann(m - 1, ackermann(m, n - 1));
8  }
```

In jedem rekursiven Aufruf wird entweder der erste Parameter kleiner oder der erste Parameter bleibt unverändert und der zweite wird kleiner. Es ist schwierig eine Abstiegsfunktion anzugeben, welche die noethersche Ordnung $(\mathbb{N}, \leq)$ der natürlichen Zahlen verwendet (wenn man nicht vorher eine ähnlich komplizierte Funktion definiert und deren Terminierung bewiesen hat). Der Terminierungsbeweis kann aber sehr einfach mit der noetherschen Ordnung $(\mathbb{N} \times \mathbb{N}, \leq_{lex})$ geführt werden:

1. Als Vorbedingung wählen wir: `requires m >= 0 && n >= 0`
2. Wenn man Überläufe nicht betrachtet, ist die Vorbedingung für rekursive Aufrufe wegen der if-Bedingungen erfüllt.

   In Zeile 5 ist $m - 1 \geq 0$, weil $m \neq 0$ und $m \geq 0$ gilt.

   In Zeile 7 gilt für den äußeren Aufruf dasselbe; außerdem ist der Rückgabewert der `ackermann`-Funktion immer mindestens 1.

   Für den inneren Aufruf in Zeile 7 gilt $n - 1 \geq 0$, weil $n \neq 0$ nach if-Bedingung.
3. Als Abstiegsfunktion wählen wir $h(m, n) = (m, n)$ und als noethersche Ordnung $(\mathbb{N} \times \mathbb{N}, \leq_{lex})$.
4. Für die rekursiven Aufrufe gilt dann:
   a) Aufruf in Zeile 5:

      $h(m - 1, 1) = (m - 1, 1) <_{lex} (m, 0) = (m, n) = h(m, n)$
   b) Äußerer Aufruf in Zeile 7:

      (Wir verwenden a als Abkürzung für `ackermann`)

      $h(m - 1, a(m, n - 1)) = (m - 1, a(m, n - 1)) <_{lex} (m, n) = h(m, n)$

c) Innerer Aufruf in Zeile 7:

$h(m, n-1) = (m, n-1) <_{lex} (m, n) = h(m, n)$

In Schritt 4 b) konnten wir hier den Abstieg zeigen, ohne den konkreten Wert von $a(m, n-1)$ zu kennen.

Zum mathematischen Begriff der Relation:
Eine Teilmenge R von $M \times N$ heißt eine (binäre) ***Relation***.

Gilt $M = N$, dann nennt man R ***homogen***.

Eine homogene Relation heißt:

- ***reflexiv***, wenn für alle $x \in M$ gilt: $(x, x) \in R$
- ***antisymmetrisch***, wenn für alle $x, y \in M$ gilt:
  wenn $(x, y) \in R$ und $(y, x) \in R$, dann $x = y$
- ***transitiv***, wenn für alle $x, y, z \in M$ gilt:
  wenn $(x, y) \in R$ und $(y, z) \in R$, dann $(x, z) \in R$

Eine reflexive, antisymmetrische und transitive homogene Relation auf $M \times M$ heißt eine ***partielle Ordnungsrelation***.
Eine Menge M mit einer Ordnungsrelation R heißt eine ***partielle Ordnung***.
Meist benutzt man Infixoperatoren wie $\leq$ (oder $\subseteq$) zur Darstellung der Relation und schreibt $x \leq y$ statt $(x, y) \in R$ und $x < y$ für $x \leq y$ und $x \neq y$.

Sei $(M, \leq)$ eine partielle Ordnung. Eine Folge $x_0, x_1, x_2, \ldots$ heißt eine ***absteigende*** Kette, wenn überall $x_i \leq x_{i+1}$ gilt. Falls sogar überall $x_i < x_{i+1}$ gilt, dann nennt man die Kette ***echt absteigend***.
Eine ***noethersche Ordnung*** ist eine partielle Ordnung, die keine unendlichen echt absteigenden Ketten enthält.
Äquivalente Bezeichnungen für Noethersche Ordnungen sind: wohlfundierte Menge, fundierte Ordnung, oder im Englischen *well-founded relation*.

2.2 Rekursion vs. Iteration

Für jede Prozedur, die ein Problem mit Hilfe von Rekursion löst, kann eine Alternativimplementierung gefunden werden, die das gleiche Problem iterativ, d.h. mit Schleifenkonstrukten löst.

```
public static int fac(int n) {
   if (n == 0) {
      return 1;
   } else {
      return n * fac (n-1);
   }
}
```

```
public static int fac(int n) {
   int result = 1;
   int i = n;
   while (i != 0) {
    result = i * result;
    i--;
   }
   return result;
}
```

Die rekursive Variante führt allerdings oft zu einfacherem und eleganterem Code, wenn man Probleme auf rekursiven Datenstrukturen löst.
Auch bei der Verwendung eines iterativen Algorithmus muss sichergestellt werden, dass die entsprechende Schleife nur endlich oft ausgeführt wird (Stichwort: "Endlosschleife"). Für Terminierungsbeweise von Schleifen kann auch die Idee der Abstiegsfunktionen verwendet werden. Auf eine Formalisierung dieser Überlegungen verzichten wir an dieser Stelle.

Manche Programmiersprachen (insbesondere funktionale Sprachen wie Haskell, Ocaml, Erlang etc.) bieten keine Sprachmittel für Schleifen an. In diesen Sprachen müssen rekursive Funktionen immer dann verwendet werden, wenn die Anzahl der Operationen nicht von vornherein beschränkt ist[6]. Andererseits gibt es einige imperative Sprachen (OpenCL, diverse C Compiler für Mikrocontroller, ältere Versionen von Fortran), die wiederum keine Rekursion als Sprachmittel zur Verfügung stellen. Programmierer müssen daher beide Varianten verstehen und anwenden können.

[6] Oft verwendet man in diesen Sprachen auch Funktionen höherer Ordnung wie `map`, `filter` und `fold`, welche von bestimmten rekursiven Anwendungsfällen abstrahieren.

Hinweise zu den Fragen

Hinweise zu Frage 1: Durch Verwendung eines ***Akkumulationsparameters***, der die Zwischenergebnisse an den rekursiven Aufruf weitergibt, kann man die Prozedur folgendermaßen anpassen:

```
public static int fac(int n) {
   return helperFac(n, 1);
}
public static int helperFac(int n, int x) {
   if (n == 0) {
      return x;
   } else {
      return helperFac(n-1, n * x);
   }
}
```

Im Terminierungsfall wird dann der Wert des Akkumulators als Ergebnis der Gesamtberechnung zurückgeliefert.