Merkblatt Algorithmen

Prof. Dr. Udo Hahn, Erik Fäßler, Johannes Hellrich, Franz Matthies

Dieses Dokument beschäftigt sich mit den grundlegenden Elementen der Algorithmisierung bzw. Programmierung. Die hier vorliegenden Beispiele sollen dem Verständnis dienen und das Lösen der Aufgaben erleichtern. Alle Punkte wurden in der Übung bereits besprochen und werden hier lediglich zum Nachsehen wiedergegeben.

1 Variablen

Den Begriff der *Variablen* (*„Veränderbare“*) kennt man aus dem Mathematikunterricht. Eine Funktion $f(x) = x^2 + 5$ enthält die Variable x. Diese Variable kann alle Werte zwischen $-\infty$ und $+\infty$ annehmen, die Funktion gibt für jedes unterschliedliche x einen Funktionswert aus. Wichtig ist, dass x unterschiedliche Werte annehmen kann, also nicht unbedingt nur einem Wert entspricht. Man weist in der Mathematik auch Werte zu. Dazu werden meist Phrasen wie „Sei x gleich 7“ verwendet. Ab diesem Zeitpunkt enthält die Variable x den Wert 7, bis ihr ein anderer Wert zugewiesen wird.

Dieses Konzept einer Variablen lässt sich direkt auf Variablen in Programmiersprachen übertragen. „Sei x gleich 7“ würde beispielsweise mit `x = 7` ausgedrückt. Das Zeichnen der Funktion $f(x) = x^2 + 5$ könnte so aussehen:

```
for (x in [−∞,... +∞])
      zeichne den Punkt (x, x^2 + 5) in das Koordinatensystem
```

Hier durchläuft x alle Werte von negativ Unendlich bis positiv Unendlich. Für jeden Wert wird der Funktionswert in das Koordinatensystem eingetragen (Natürlich würde das obenstehende Programm in Wahrheit niemals fertig werden!).

In Programmiersprachen werden Variablen häufig schlicht als „Behälter“ verwendet, um sich Zahlen (oder Zeichenketten und weiteres) zu „merken“.

Beispiele:

Einfache Zuweisungen: Die Zahl rechts neben dem Pfeil wird der Variable zugewiesen

```
x <- 5
y <- 7
z <- 8
```

Einfache Rechenschritte (mit den Variablenwerten von oben):

```
x <- 3 * z          // x beinhaltet nun den Wert 3 * z = 3 * 8 = 24
x <- x / 2          // x beinhaltet seinen eigenen Wert durch zwei geteilt:
z <- x + y          // z = x + y = 12 + 7 = 19
```

1.1 Strings

Wir arbeiten oft mit Zeichenketten/Strings, also Folgen von Buchstaben oder anderen Zeichen. Diese werden durch Anführungszeichen kenntlich gemacht. Dabei sind Strings die eine Zahl enthalten von der Zahl selbst zu unterscheiden, im folgenden Beispiel etwa ist der Wert von y ungleich dem von z.

```
x <- "Baum"
y <- "8"
z <- 8
```

Für Strings gilt, dass wir sie durch + konkatinieren (verbinden) können, im folgenden nimmt z den Wert "12" an.

```
x <- "1"
y <- "2"
z <- x + y
```

Wir können sie auch substrahieren, wobei hier der linke String vom Anfang des rechten entfernt wird, es gilt somit:
`"x" - "xy" = "y"`
Zusätzlich können wir auch ihre Länge, also die Anzahl der Zeichen aus denen sie bestehen abfragen. Dazu benutzen wir entweder eine Funktion `length(String)` (unten mehr zu Funktionen) oder Betragsstriche `||String||`.

2 Programmfluss

Beim Lesen bzw. Schreiben von Programmen im Pseudecode ist zu beachten, dass der Programmfluss stets von oben nach unten, Zeile für Zeile statt findet.

Beispiele:

Es folgen zwei Sequenzen von Anweisungen. Dabei sind die Anweisungen selbst die gleichen, allerdings ist ihre Reihenfolge vertauscht (zur besseren Erklärbarkeit wurden Zeilennummern hinzugefügt).

Sequenz 1:

```
1. x <- 4
2. y <- 9
3.
4. y <- y + x
5. x <- x * 2
6. print y
Ausgabe 13
```

Sequenz 2:

```
1. x <- 4
2. y <- 9
3.
4. x <- x * 2
5. y <- y + x
6. print y
Ausgabe 17
```

Am Ende von Sequenz 1 beinhaltet die Variable y den Wert 13, x den Wert 8. Am Ende von Sequenz 2 beinhaltet y jedoch den Wert 17, während x wieder eine 8 repräsentiert. Der – vielleicht offensichtliche aber dennoch erwähnenswerte – Grund für den Unterschied liegt in der Reihenfolge der Anweisungen. In der Zeile, in der y ein neuer Wert zugewiesen wird (4. bzw. 5. Zeile) ist es wichtig, welchen Wert x zu *diesem Zeitpunkt* enthält. Wird zuerst die Zweisung an y getätigt und dann x verdoppelt (erste Sequenz), so bleibt die Zuweisung an y von der Verdopplung unberührt. Bei der y-Zuweisung in der zweiten Sequenz wurde x bereits verdoppelt und geht somit auch verdoppelt in y ein.

3 Blöcke

Die *Kontrollstrukturen* `if / else`, `for` und `while` werden verwendet, um den *Programmfluss zu ändern*. Sie werden also immer dann gebraucht, wenn nicht einfach jede Zeile von oben nach unten nacheinander ausgeführt werden soll. Die Strukturen werden weiter unten noch einmal näher beschrieben. In diesem Abschnitt geht es darum zu verstehen, wie Anweisungen zu *Blöcken* zusammengefasst werden. So können mehrere Anweisungen auf einmal aus dem normalen Programmfluss ausgenommen werden.

Jede `if`-, `for`- oder `while`-Anweisung bezieht sich auf einen Block von Anweisungen. Es gehören die Anweisungen zu einem Block, die direkt unter der entsprechenden Anweisung *eingerückt* sind.

Beispiel:

```
1. x <- <eine Zahl, die von Programmaufruf zu Programmaufruf unterschiedlich sein kann>
2. y <- 0
3. if x > 3
4.      x <- 6
5.      x <- x + 1
6.      y <- x - 3
7.      y <- y * y + 4
8. y <- y - 3
```

Die Anweisungen der Zeilen 4 bis 7 gehören als Block zu der `if`-Struktur in Zeile 3. Der Programmfluss gestaltet sich wie folgt:

Zeile 1 wird ausgeführt.
Zeile 2 wird ausgeführt.
Zeile 3 wird ausgeführt (der Test, ob x größer als 3 ist).
Die Zeile 4, 5, 6 und 7 werden ausgeführt, falls x größer als 3 ist. Diese Zeilen werden *immer* zusammen ausgeführt oder überhaupt nicht ausgeführt.
Zeile 8 wird ausgeführt.

Achtung: Blöcke dürfen ineinander geschachtelt werden:

```
1. x <- <eine Zahl, die von Programmaufruf zu Programmaufruf unterschiedlich sein kann>
2. y <- 0
3. if x > 3
4.      repeat
5.            x <- x + 1
6.      until x > 5
7.      y <- x - 3
8. y <- y - 3
```

4 if / else

Die if / else - Struktur („wenn / ansonsten“) bietet eine Möglichkeit, um den normalen Programmfluss eines Programms anzupassen. Die Struktur stellt die Ausführung *ihres Blocks* unter eine Bedingung. `if x > 3` bedeutet „wenn die Variable x den Wert 3 überschreitet – und nur dann – wird der folgende Anweisungsblock ausgeführt“. Ist das nicht der Fall, werden die entsprechenden Anweisungen *übersprungen*, wie im vorigen Abschnitt erklärt wurde. Ein `if` kann dabei ohne `else` vorkommen:

```
if x > 3
     x <- x + 5
     print x
<weitere Anweisungen>
```

Nur wenn x größer ist als 3 wird die Variable um 5 erhöht und anschließend ausgegeben. Ist die Variable kleiner oder gleich 3 passiert *nichts*. Alle Anweisungen nach dem `if`-Block werden auf jeden Fall ausgeführt (es sei denn natürlich, dort finden sich weitere Kontrollstrukturen).

Mit der `else`-Anweisung kann mitgeteilt werden, was passieren soll, falls die `if`-Bedingung *nicht* zutrifft.

```
if x > 3
     x <- x + 5
     print x
else
     x <- x + 10
     print "x wurde um 10 erhöht"
<weitere Anweisungen>
```

Falls x nun also nicht größer als 3 ist, wird eine 10 hinzuaddiert und eine Meldung ausgegeben, dass die Erhöhung um 10 statt gefunden hat. Der Block unter dem `if` wird *nicht* ausgeführt, wohl aber der Block unter dem `else`. Die „weiten Anweisungen“ sind nach wie vor unberührt von der `if / else`-Struktur.

5 Schleifen

Schleifen werden eingesetzt, um einen Anweisungsblock mehrmals zu *wiederholen*. Die `for`Schleife – hier in Form der for-each-Schleife – nimmt nacheinander jedes Element einer Liste oder jeden Buchstaben eines Strings und belegt eine Variable damit. Für jeden Wert der Variable wird der Anweisungsblock der Schleife ausgeführt:

```
summe <- 0
for i in [1,2,3,4,5]
      summe <- summe + i
<weitere Anweisungen>
```

Die *Schleifenvariable* i nimmt jeden (ganzzahligen) Wert in der Liste [1,2,3,4,5] an. Für jeden dieser Werte wird der Schleifenblock (auch: Schleifenkörper) ausgeführt. Ersetzt man im obigen Schleifenkörper in jedem Durchlauf das i durch den aktuellen Wert des Durchlaufs, erhält man die Anweisungen

```
summe <- summe + 1
summe <- summe + 2
summe <- summe + 3
summe <- summe + 4
summe <- summe + 5
```

Es wird also die Summenformel von 5 berechnet. Wichtig ist sehen, dass die „weiteren Anweisungen“ erst nach Beendigung der Schleife ausgeführt werden. Der Programmfluss dreht sich so lange „im Kreis“ und wiederholt immer wieder den Schleifenkörper, bis i den Wert 6 annehmen würde. Erst dann wird die Schleife verlassen.

Die zweite Schleifenform, die wir hier verwenden, ist die `while`-schleife. Sie besitzt keine Zählvariable sondern lediglich eine Bedingung. Der Schleifenkörper einer `while`-Schleife wird so lange wiederholt, wie die Bedingung zutrifft (ähnlich wie die `if`-Anweisung, mit dem Unterschied, dass bei `if` der Anweisungsblock nur ein einziges mal ausgeführt wird).

Beispiel:

```
summe <- 1
while summe < 7 do
      summe <- summe + 2
<weitere Anweisungen>
```

Es ist wichtig zu sehen, dass diese Schleife nur deshalb irgendwann aufhört, weil die Bedingung schließlich nicht mehr zutreffen wird. Dies geschieht durch die Erhöhung der Variable `summe`, deren Wert in der Bedingung abgefragt wird. Mit anderen Worten: Damit eine `while`-Schleife jemals beendet wird, muss der Schleifenkörper Anweisungen enthalten, die die Schleifenbedingung irgendwann – am besten, wenn die Arbeit der Schleife getan ist – ungültig machen. Auch hier gilt: So lange die Schleife nicht beendet wurde, werden die „anderen Anweisungen“ nicht ausgeführt. Das heißt, so lange die Schleifenbedingung wahr ist, werden die Anweisungen nach der Schleife nicht betrachtet.

6 Schachtelung von Anweisungsblöcken

Kontrollstrukturen können ohne weiteres geschachtelt werden. Das heißt, eine Kontrollstruktur befindet sich im Anweisungsblock einer anderen Struktur.

Beispiel:

```
for i in [1,2,3,4,5,6,7,8,9,10]
      if i = 5 then
            print "Die fünf wurde erreicht!"
      else
            print "Das ist keine fünf."
```

Für jedes i (also $1, 2, 3, \ldots, 10$) wird der Schleifenkörper ausgeführt. dieser besteht aus einer `if / else`-Struktur. Zum Verständis kann es helfen, den Schleifenblock losgelöst zu betrachten:

```
if i = 5 then
      print "Die fünf wurde erreicht!"
else
      print "Das ist keine fünf."
```

„Ist die Variable i gleich 5, so gib das aus, ansonsten gib aus, dass keine 5 vorliegt“. Man muss sich klar machen, dass diese Anweisungen einfach für jedes i zwischen 1 und 10 ausgeführt werden, wenn sie als Anweisungsblock der obigen `for`-Schleife vorkommen.

7 Arbeiten mit Pseudocode

Wir arbeiten hier mit Pseudocode und keiner real existierenden Programmiersprache. Das hat den Vorteil, das man Algorithmen nah an der eigenen Intuition schreiben kann. So wäre auch die Anweisung

```
if x größer als 5 then
      print "Die Variable ist größer als 5!"
```

gültig. Ein Test, ob eine Variable durch 7 teilbar ist oder nicht könnte mit

```
if x % 7 = 0 then
      print "Die Variable durch 7 teilbar!"
```

als auch mit

```
if x ist durch 7 teilbar then
      print "Die Variable durch 7 teilbar!"
```

ausgedrückt werden. Hauptsache ist Verständlichkeit sowie die Lösung des Hauptproblems durch algorithmische Strukturen. Falls beispielsweise die Länge einer Liste bestimmt werden muss, um eine höhere Aufgabe zu lösen, so kann geschrieben werden

```
liste <- [1, 4, 23]
länge <- length(liste)    // oder "länge von liste", "len(liste)",...
```

Es kann sich so auf die Hauptaufgaben fokussiert werden.

8 Funktionen

Funktionen erlauben es uns Abläufe, die wir identisch oder sehr ähnlich an verschiedenen Stellen brauchen, an einem Ort zu sammeln. Wir definieren eine Fuktion mit ihrem Namen, ihrem Input (auch Parameter oder Argumente genannt) und Output (auch Rückgabewert genannt) einmal, können sie danach aber immer wieder aufrufen. Die folgende Beispielsfuntion gibt als Output den zu sich selbst addierten Input zurück:

```
def beispielsfunktion(↓input, ↑output)
      output ← input + input
```

Sie gibt beispielsweise beim Aufruf `beispielsfunktion("string")` den Wert `"stringstring"` zurück.
Stehen mehrere Funktionsaufrufe in einer Zeile werden sie von links nach rechts bearbeitet, wenn mehrere Funktionsaufrufe ineinander geschachtelt sind werden sie von innen nach außen bearbeitet – das Vorgehen entspricht somit dem beim Lösen einer Gleichung wie $x = 1 * (2 + 2)$. Beispielsweise würde bei `funktion1(funktion2("string"))` zuerst `funktion2("string")` ausgeführt und dann `funktion1` mit deren Ergebnis.
Achtung:
Wird eine Funktion später wieder aufgerufen (oder eine rekursive Funktion mehrfach nacheinander), so sind die Variablen völlig unabhängig voneinander. Somit gilt: `beispielsfunktion(1) + beispielsfunktion(1)` $= 4$ und nicht etwa 6.