# Einführung in die Programmierung mit Snap!

Fritz Hasselhorn

27. Februar 2023

# Inhaltsverzeichnis


**1. Grundlagen der Programmierung 6**
1.1. Kontrollstrukturen 6
1.1.1. Anweisung 6
1.1.2. Sequenz 7
1.1.3. Verzweigung 7
1.1.4. Schleifenarten 7
1.2. Blöcke in Snap! 11
1.2.1. Command 11
1.2.2. Reporter 11
1.2.3. Prädikat 12
1.2.4. Hut-Blöcke 12
1.2.5. C-förmige Blöcke 12
1.3. Variable in Snap! 13
1.3.1. Globale Variable 13
1.3.2. Parameter 13
1.3.3. Skriptvariable 15
1.3.4. Verwendung der Variablenarten 15
1.4. Aufgaben 16

**2. Werkzeuge 17**
2.1. Notation von Snap! 17
2.2. Struktogramm 18
2.3. Tracetabelle 21

**3. Grafik 26**
3.1. Koordinaten 26
3.2. Grafikbefehle 26
3.3. Die kleine Stadt 27
3.4. Galgen zeichnen 31
3.5. RGB-Farbmodell 31
3.6. Farbverlauf 32
3.7. Farbige Kreise 34
3.8. Der fliegende Ball im Netzwerk 35
3.9. Regeln für Zeichenblöcke 38
3.10. Aufgaben 38

**4. Mathematische Algorithmen 39**
4.1. Datentyp Zahlen 39
4.2. Rund um MOD 40
4.2.1. Zählen mit MOD 40
4.2.2. DIV - die ganzzahlige Division 41
4.2.3. Paritätsbit 41
4.2.4. Größter gemeinsamer Teiler 42

4.3. Wahrheitswert . . . . . 43
4.4. Aufgaben . . . . . 43

**5. Dualzahlen** **46**
5.1. Grundlagen . . . . . 46
5.2. Dezimal nach Dual . . . . . 47
5.3. Dual nach Dezimal . . . . . 49
5.4. Rechnen mit Dualzahlen . . . . . 50
5.5. Wechselgeldautomat . . . . . 51
5.6. Hexadezimalzahlen . . . . . 52
5.7. Aufgaben . . . . . 54

**6. Umgang mit Zeichenketten** **56**
6.1. ASCII-Code . . . . . 56
6.2. Buchstaben . . . . . 56
6.3. Zeichenketten . . . . . 57
6.4. Grundrezept Operationen mit Zeichenketten . . . . . 58
6.5. Lexikographie . . . . . 59
6.6. Galgenraten . . . . . 60
6.7. Zeichenketten mit Listenbefehlen . . . . . 61
6.8. Aufgaben . . . . . 62

**7. Sortieren** **63**
7.1. Listen . . . . . 63
7.2. Grundrezept InsertionSort . . . . . 64
7.3. Grundrezept SelectionSort . . . . . 66
7.4. Bubblesort . . . . . 66
7.5. Aufgaben . . . . . 67

**8. Verschlüsselungsverfahren** **69**
8.1. Kryptosysteme . . . . . 69
8.2. Caesar . . . . . 71
8.3. Skytale . . . . . 72
8.4. Häufigkeit . . . . . 74
8.5. Balkendiagramm . . . . . 75
8.6. Aufgaben . . . . . 77

**9. Häufige Fehler** **78**
9.1. Wertzuweisungen . . . . . 78
9.2. Händische Inkrementierung der FOR-Schleife . . . . . 78
9.3. FOR-Schleife läuft rückwärts . . . . . 78
9.4. Doppelkreuz in der Kopfzeile von Hand . . . . . 79
9.5. Verstoß gegen die Datenkapselung . . . . . 80
9.6. Addieren mit dem ADD-Block . . . . . 80
9.7. Parameter . . . . . 80
9.8. Alleinstehender Reporter . . . . . 80
9.9. Verwendung der deutschen Sprache . . . . . 80
9.10. Struktogramm-Schreibweisen im Quelltext . . . . . 81
9.11. Kopieren von Listen . . . . . 81

**10. Liste der Snap!-Blöcke** **82**
10.1. Grundlagen der Programmierung . . . . . 82
10.2. Werkzeuge . . . . . 82
10.3. Grafik . . . . . 82
10.4. Mathematische Algorithmen . . . . . 83
10.5. Dualzahlen . . . . . 84
10.6. Umgang mit Zeichenketten . . . . . 84
10.7. Sortieren . . . . . 84
10.8. Verschlüsselungsverfahren . . . . . 84
10.9. Blöcke aus NetsBlox . . . . . 84

**11. Übungsklausur** **85**

**A. ASCII-, Hexadezimal- und Dualzahlen-Tabelle** **88**

# 1. Grundlagen der Programmierung

In diesem Kapitel lernen Sie

- wie sich Algorithmen aus den Grundbausteinen Anweisung, Sequenz, Schleife und Verzweigung zusammensetzen.
- wie man Algorithmen in Form von Struktogrammen darstellt,
- welche Arten von Schleifen es gibt,
- was eine Zählschleife leistet und weshalb man die FOR-Schleife benutzt,
- welche Arten von Blöcken Snap! verwendet und
- wie man zwischen lokalen (Skript-) und globalen Variablen und Parametern unterscheidet.

## 1.1. Kontrollstrukturen

Als Kontrollstrukturen bezeichnen wir in der Informatik Anweisungen, die den Ablauf eines Programms steuern. Sie dienen der besseren Lesbarkeit des Programmtextes und lösen die früher häufig verwendeten Sprungbefehle (GOTO) ab.[1] Ein Sprungbefehl bedeutet, dass die Programmausführung an der Sprungadresse fortgesetzt wird. Das Konzept stammt aus der Assemblerprogrammierung und ist eine wichtige Ursache dafür, dass Assemblerprogramme nur schwer lesbar sind.

Böhm und Jacopini konnten 1966 zeigen, dass die drei Kontrollstrukturen Sequenz, Verzweigung und Schleife ausreichen, um jeden Quelltext in ein GO-TO-freies Programm umzuschreiben.[2] Dazu wird der Quelltext in Blöcke (Operationen, Funktionen) zerlegt. Deshalb beschränken wir uns auf diese drei Kontrollstrukturen und Anweisungen im Sinne des Kerncurriculums Informatik.

[1] Edsger W. Dijkstra, "Go To Statement Considered Harmful", Communications of the ACM, März 1968

[2] Bohm, Corrado; Giuseppe Jacopini, Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules". Communications of the ACM, Mai 1066

Um unabhängig von einer konkreten Programmiersprache die logische Struktur von Operationen darstellen zu können, entwickelten Isaac Nassi und Ben Shneiderman 1972/1973 die nach ihnen benannten Nassi-Shneiderman-Diagramme.[3] Sie werden auch als **Struktogramme** bezeichnet.

### 1.1.1. Anweisung

Wenn das Kerncurriculum Informatik von Anweisungen spricht, meint es einen eingeschränkteren Begriff als den im ersten Abschnitt erwähnten erweiterten Begriff von Anweisung, der die Kontrollstrukturen mit umfasst. Hier ist eher an Wertzuweisungen (einer Variable wird ein Wert zugewiesen) gedacht. Snap! verwendet für Anweisungen in diesem engeren Sinn den Begriff **Command**. Damit ist gemeint, dass der Computer einen Befehl ausführt, z.B. eine Bewegung oder eine sonstige Aktion, oder einen anderen Block aufruft. In Snap! haben Blöcke, die als Anweisung ausgeführt werden können, die Form eines Puzzleteils, um anzudeuten, dass diese Anweisungen zu komplexeren Blöcken (im nächsten Abschnitt als Sequenzen bezeichnet) zusammengesetzt werden können.

Abbildung 1.1.: Anweisung

[3] normiert in der DIN 66261

## 1.1.2. Sequenz

Eine Sequenz besteht aus mehreren Anweisungen, die nacheinander von oben nach unten ausgeführt werden. Die einzelnen Anweisungen einer Sequenz werden wie Puzzleteile zusammengesteckt. Ovale Reporter können nicht in einer eigenen Zeile stehen, sondern nur in Commands verwendet werden.

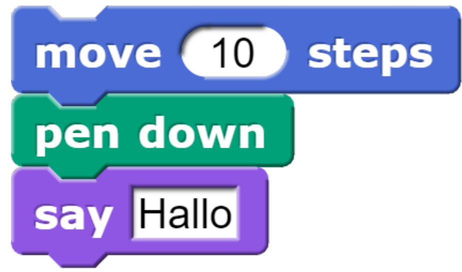



Abbildung 1.2.: Sequenz

## 1.1.3. Verzweigung

Abbildung 1.3.: Einseitige Verzweigung

Eine Verzweigung (auch Auswahl oder Selektion genannt) besteht aus einer Bedingung und bis zu zwei Codeabschnitten, die in Abhängigkeit von der Bedingung ausgeführt werden.

Die Bedingung ist dabei ein Predicate (Theoretisch könnte man auch einen Wahrheitswert (wahr/falsch) in das sechseckige Feld einsetzen, aber damit wäre ein Weg vorgegeben und somit keine Verzweigung mehr möglich).

Die einseitige Verzweigung oben hat nur einen Codeabschnitt, der ausgeführt wird, falls die Bedingung zutrifft. Falls die Bedingung nicht zutrifft, wird der Codeabschnitt übersprungen.

Die zweiseitige Verzweigung hat zwei Codeabschnitte. Der eine wird ausgeführt, falls die Bedingung zutrifft, der andere, falls sie nicht zutrifft.

## 1.1.4. Schleifenarten

Schleifen dienen der wiederholten Ausführung von Befehlen. Wir unterscheiden drei Arten von Schleifen: vorprüfende Schleifen, nachprüfende Schleifen und Zählschleifen.

### Vorprüfende Schleife

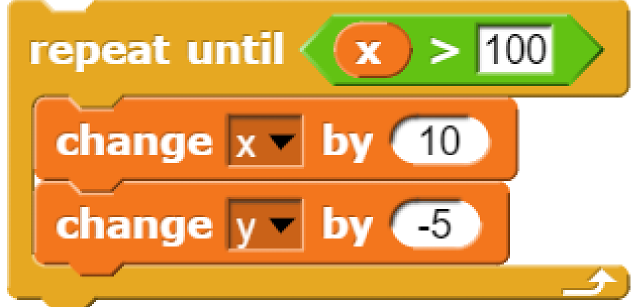



Abbildung 1.4.: Vorprüfende Schleife

Eine vorprüfende Schleife, auch `WHILE`-Schleife genannt, prüft zunächst eine Bedingung, bevor sie Code ausführt, daher auch der Name. Wird die Bedingung erfüllt, so wird der Codeabschnitt innerhalb der Schleife ausgeführt. Anschließend wird wiederum die Bedingung überprüft und, falls zutreffend, der Codeinhalt innerhalb der Schleife ausgeführt, usw. An dieser Stelle muss der Programmierenden darauf achten, dass die Bedingung auch erfüllt wird. Sonst bleibt der Computer in einer Endlosschleife gefangen und kann den Rest des Programms nicht mehr ausführen. Die Bedingung im Kopf der Schleife regelt, wie lange die

Ausführung des Codes wiederholt wird. Wir sprechen deshalb von einer Wiederholungsbedingung.

In vielen Programmiersprachen, z.B. Delphi, wird die vorprüfende Schleife mit dem `WHILE (Wiederholungsbedingung) DO ...` bezeichnet. Daher kommt die Bezeichnung `WHILE`-Schleife. Snap! verfügt standardmäßig über keinen `WHILE`-Block. Bei Bedarf kann der Block aber über **import libraries (Iteration, composition)** nachgeladen werden.

Der `REPEAT UNTIL (Bedingung)`-Block in Snap! ist zwar auch ein vorprüfender Block, aber er prüft nicht die Wiederholungsbedingung, sondern die Abbruchbedingung. Ist die Abbruchbedingung zu Beginn erfüllt, so wird der Code in der Schleife nicht ausgeführt. Um aus einer Wiederholungsbedingung eine Abbruchbedingung zu machen, muss die Bedingung durch ein vorangestelltes `NOT` verneint werden.

Wichtig dabei ist, dass bei zusammengesetzten Bedingungen die komplette Wiederholungsbedingung eingeklammert wird, damit sich die Verneinung auf die ganze Bedingung bezieht. Will man eine zusammengesetzte Bedingung auflösen, dann sind die Regeln von De Morgan zu beachten (den Beweis für die Regeln findet man bei „Gesetze der Schaltalgebra"):

$$\overline{a \vee b} = \overline{a} \wedge \overline{b}$$

$$\overline{a \wedge b} = \overline{a} \vee \overline{b}$$

Diese Regeln besagen, dass die Verneinung einer `AND`-Verknüpfung auch dargestellt werden kann, indem die beiden Einzelbedingungen verneint und dann mit `OR` verknüpft werden. Die Verneinung einer `OR`-Verknüpfung kann auch dargestellt werden, indem die beiden Einzelbedingungen verneint und dann mit `AND` verknüpft werden.

### Nachprüfende Schleife

Der Inhalt einer nachprüfenden Schleife wird in jedem Fall einmal ausgeführt. Erst nach der ersten Wiederholung erfolgt die Prüfung einer Abbruchbedingung. Anschließend folgen weitere Wiederholungen, bis die Abbruchbedingung erfüllt wird. Snap! verfügt standardmäßig über keine nachprüfende Schleife. Allerdings steht eine solche unter **Libraries** im ersten Eintrag **Iteration, composition** zur Verfügung und kann bei Bedarf importiert werden. Bei vielen älteren Programmiersprachen heißt die nachprüfende Schleife `repeat Schleifeninhalt until Abbruchbedingung`. Sie ist von der vorprüfenden `repeat until Abbruchbedingung Schleifeninhalt` zu unterscheiden.

Wir können eine nachprüfende Schleife auch mit Hilfe unserer üblichen `repeat until`-Schleife in Snap!-Code umsetzen. Betrachten wir dazu ein kleines Beispiel. Wir wollen zwei Zufallszahlen `z1, z2` auswählen, die verschieden sein sollen. Wir wählen also zunächst die Zufallszahl `z1` und dann die Zufallszahl `z2`, wobei wir die Wahl von `z2` so lange wiederholen, bis `z2` von `z1` verschieden ist. Wir stellen vor unseren `repeat until`-Block also eine Anweisung zur Auswahl von `z2`. Dann folgt die `repeat until`-Schleife, die prüft, ob die Abbruchbedingung bereits erfüllt ist, und nur dann eine weitere Wiederholung durchführt, falls die Abbruchbedingung noch nicht erfüllt ist. Sind die beiden Zufallszahlen bereits verschieden, dann wird der Inhalt der Schleife nicht mehr ausgeführt.

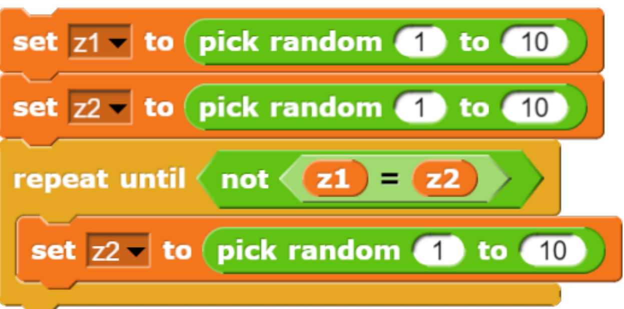



Abbildung 1.5.: Nachprüfende Schleife in Snap!

Allgemein gilt: Will man eine nachprüfende Schleife ohne Import aus den Libraries direkt implementieren, so muss zunächst der Quelltext einmal aufgeführt werden, dann erfolgt mit `repeat until` die Prüfung der Abbruchbedingung und darin eingeschachtelt erneut der Quelltext für ggf. notwendige weitere Wiederholungen:

### Zählschleife

Eine Sonderform der vorprüfenden Schleife ist die Zählschleife. Der wesentliche Unterschied einer Zählschleife zu einer einfachen vorprüfenden Schleife wie `repeat <anzahl>` bzw. `repeat until <Bedingung>` besteht darin, dass die Zählschleife zusätzlich über eine Zählvariable verfügt. Eine Zählvariable verändert nach jeder Wiederholung ihren Wert, meist um Eins. Sie wird häufig benutzt, um nacheinander auf die Buchstaben eines Wortes oder auf die Elemente einer Liste zuzugreifen.

Eine Zählschleife leistet drei Dinge:

- Zunächst wird die Zählvariable auf den Startwert gesetzt. In der Fachsprache sagen wir: Die Zählvariable wird **initialisiert**.
- Dann wird der Codeabschnitt im Inneren der Schleife ausgeführt. Anschließend wird die Zählvariable (in der Regel) um Eins erhöht. Wir sagen: Die Zählvariable wird **inkrementiert**.
- Es schließt sich die nächste Wiederholung an mit Ausführung und Inkrementierung usw. Wenn die Zählvariable den Endwert erreicht hat, wird der Code noch ein Mal ausgeführt und die Zählvariable inkrementiert. Dann wird die Schleife verlassen. Die Zählschleife sorgt also für die **richtige Anzahl an Wiederholungen**.

Zu beachten ist dabei die Anzahl der Wiederholungen. Nehmen wir an, eine Zählschleife zählt von 2 bis 5. Dann ist 2 der Startwert und 5 der Endwert der Zählvariable. Es sind insgesamt vier Wiederholungen der Schleife notwendig: eine mit 2, eine mit 3, eine mit 4 und mit 5. Die Differenz von End- und Startwert beträgt aber nur 3. Weil sowohl ein Durchgang mit dem Startwert auch auch ein Durchgang mit dem Endwert verlangt wird, beträgt die Anzahl der Wiederholungen **Endwert−Startwert+1**.

Programmiert man eine Zählschleife unter Snap! von Hand, so muss man drei Dinge tun:

1. Die Zählvariable (hier i) auf den Startwert setzen (sie initialisieren)
2. Die richtige Anzahl von Wiederholungen einstellen (Endwert − Startwert + 1)
3. Am Ende der Wiederholungsschleife die Zählvariable inkrementieren (erhöhen), damit der nächste Durchgang mit dem neuen Wert durchgeführt wird.

```
set i to 1
repeat (n - 1) + 1
    replace item i of Liste with i × i
    change i by 1
```

Abbildung 1.6.: händische Zählschleife

Achtung! Verwendet man statt der `REPEAT`-Schleife unter Snap! eine `REPEAT-UNTIL`-Schleife, dann muss man die obere Grenze so wählen, dass der Codeabschnitt auch mit dem Endwert noch durchgeführt wird. Statt

```
set i to 1
repeat until i = n
    replace item i of Liste with i × i
    change i by 1
```

wäre richtig

```
set i to 1
repeat until i > n
    replace item i of Liste with i × i
    change i by 1
```

Abbildung 1.7.: Zählschleife richtig

Zählschleifen sind weit verbreitet. Deshalb gibt es in den meisten Programmiersprachen den Typ der `FOR`-Schleife, der alle drei Funktionen (initialisieren, ausführen und inkrementieren) in einem Befehl zusammenfasst. Auch Snap! verfügt im Reiter **Control** über einen `FOR`-Befehl:

Der `FOR`-Block ist besonders benutzerfreundlich gestaltet. Ist der Startwert kleiner als der

Endwert, zählt der Block vorwärts vom Startwert bis zum Endwert. Ist der Startwert größer als der Endwert, zählt der Block rückwärts bis zum Endwert.

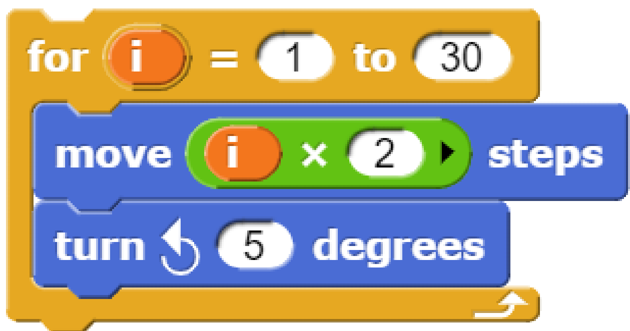



Abbildung 1.8.: FOR-Schleife

In schriftlichen Lernkontrollen, angefangen vom Kurztest bis zur Abiturprüfung, und beim Programmieren wird dringend die Verwendung des `FOR`-Befehls angeraten. Spätestens wenn man nicht nur eine einzelne Zählschleife verwendet, sondern mehrere ineinander geschachtelte, steigt die Anzahl der möglichen Fehlerquellen an. Zählschleifen von Hand sind unter Stress nicht mehr schülersicher. Mit dem `FOR`-Block vermeidet man viele Fehlerquellen.

Es gibt allerdings eine Besonderheit: Wenn die `FOR`-Schleife eine Liste entlang geht und während des Durchgangs Elemente gelöscht werden, dann muss man die Inkrementierung der Zählvariable händisch rückgängig machen, damit nicht die Elemente nach den gelöschten übersprungen werden, oder man verwendet eine händische Zählschleife.

Abbildung 1.9.: FOR-Schleife mit händischer Inkrementierung der Zählvariable

Wenn die Zählvariable durch den Programmierer verändert wird, werden bestimmte Wiederholungen weggelassen oder mehrfach ausgeführt. Der Block „`FOR`-Schleife mit händischer Inkrementierung der Zählvariablen" schreibt nicht die Zahlen von 1 bis 10 in myList, sondern 1, 3, 5, 7, 9. Mit solchen Veränderungen der Zählvariablen muss man vorsichtig umgehen. Wird z.B. die letzte Zeile durch `change i by -1` ersetzt, entsteht eine Endlosschleife.

Abbildung 1.10.: FOR-Schleife mit händischer Dekrementierung

Das Skript im zweiten Beispiel dient dazu, doppelte Zahlen aus einer sortierten Liste zu löschen. Dazu wird die Liste mit einer Zählschleife durchgegangen und das aktuelle Element mit dem vorherigen verglichen. Sind sie gleich, dann wird das aktuelle Element gelöscht. Dadurch rücken alle folgenden Elemente einen Platz vor. Die Zählvariable i wird am Ende der Schleife automatisch um 1 hochgezählt. Wird jetzt z.B. das 5. Element gelöscht, rückt das 6. an seine Stelle und das 7. rückt an Stelle 6. Wenn die `FOR`-Schleife ohne Korrektur angewandt wird, wird das vorgerückte 6. Element nicht mit dem 4. verglichen. Deshalb nach jedem Löschvorgang die Zählvariable von Hand dekrementiert und dann automatisch inkrementiert. Beides hebt sich gegenseitig auf. Das sorgt dafür, dass trotz des Löschens kein Element übersprungen wird. Die zusätzliche Schleife `if i` $\leq$ `length of data` verhindert, dass die Schleife ausgeführt wird, wenn die Zählvariable größer ist als die Länge der Liste. Dies kann passieren, da Elemente aus der Liste gelöscht werden.

Der fertige `FOR`-Block in Snap! zählt übrigens sowohl aufwärts wie abwärts. In der Regel setzen wir den Block so ein, dass der Startwert kleiner ist als die Endwert. In diesem Fall zählt der Block aufwärts. Ist allerdings der Startwert kleiner als der Endwert, dann wird die Zählvariable in jedem Durchgang nicht erhöht,

sondern um Eins verringert.

Die Zählvariable einer `FOR`-Schleife darf übrigens nicht für den Start- oder Endwert eingesetzt werden. Allerdings kann bei geschachtelten Schleifen die Zählvariable der äußeren Schleife als Start- oder Endwert der inneren Schleife eingesetzt werden.

## 1.2. Blöcke in Snap!

Snap! kennt fünf Arten von Blöcken:

1. Command-Blöcke
2. Reporter-Blöcke (im Quelltext erkenntlich am Typ des Ergebnisses in der Kopfzeile)
3. Prädikat-Blöcke (im Quelltext erkenntlich am Typ Wahrheitswert in der Kopfzeile)
4. Hut-Blöcke (erkenntlich an der geschwungenen Hut-Form)
5. C-förmige Blöcke (im Quelltext sollte die innere C-Kurve markiert werden)

### 1.2.1. Command

Command-Blöcke sind gestaltet wie ein Puzzlestück. Sie dienen zur Ausführung von Anweisungen.

Einige wenige Command-Blöcke wie Report- und Stop-Script haben abweichend vom üblichen Design eine glatte Unterseite. Hier können unten keine weiteren Blöcke angedockt werden.

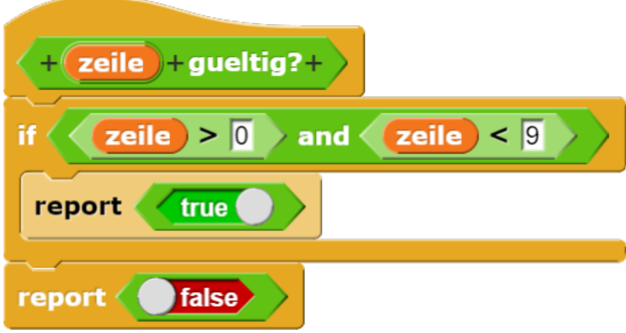



Abbildung 1.11.: Verwendung Report-Block

Ein Report-Befehl liefert den angegebenen Rückgabewert und beendet dann den aktuellen Block. Damit kann ein Block an mehreren Stellen verlassen werden:

Der Prädikatsblock prüft, ob der Wert von Zeile größer ist als 0 und kleiner als 9, also, ob er zwischen 1 und 8 liegt. Ist dies der Fall, so liefert er den Wert „true“ und beendet damit den Block. Der zweite Report-Befehl wird nicht mehr ausgeführt. Ist die Bedingung nicht erfüllt, wird der Codeabschnitt innerhalb der Verzweigung übersprungen und der zweite Report-Befehl ausgeführt.

Abbildung 1.12.: Stop-Block

Der `Stop`-Block hat vier zusätzliche Menüoptionen. In der Grundform `stop all` wird er benutzt, um alle Skripte zu stoppen. Die Option `stop this script` kann ein Skript anhalten. Die Option `stop this block` wird innerhalb der Definition eines benutzerdefinierten Blocks verwendet, um nur diesen benutzerdefinierten Blocks zu stoppen und das Skript fortzusetzen, das ihn aufgerufen hat. `stop all but this skript` kann z.B. am Ende eines Spiels verwendet werden, um alle Spielfiguren anzuhalten, aber dieses Skript weiterlaufen zu lassen, um dem Benutzer den Endstand zu liefern. Die letzte Menüoptionen `stop other scripts in sprite` hält alle anderen Skripte innerhalb des Sprites an.

### 1.2.2. Reporter

Ein Reporter liefert einen Rückgabewert, der z.B. einer Variablen zugewiesen oder in einer Bedingung ausgewertet werden kann.

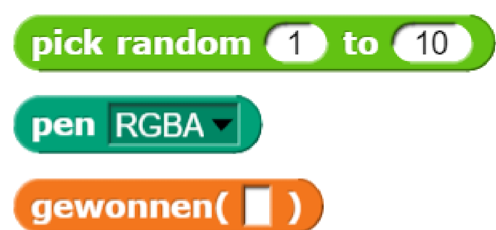



Abbildung 1.13.: Beispiele für Reporter-Blöcke

Ein Reporterblock kann nie allein in einer Programmzeile stehen, er muss in einen anderen Block eingesetzt werden. In Snap!

wird dies dadurch deutlich gemacht, dass Reporter durch einen ovalen Block symbolisiert werden, der nicht zu den Puzzleteilen der Commandblöcke zusammen geklickt werden kann, sondern dort nur in die entsprechenden Lücken eingefügt werden können. Beispiel: `join (Hallo, World)` liefert eine Zeichenkette, die aus den beiden Teilen zusammengesetzt ist. Der Reporter enthält aber keine Wertzuweisung, mit der das Ergebnis gespeichert wird.

Einen selbstgeschriebenen Reporter kennzeichnen wir im Quelltext, indem wir einen Doppelpunkt und den entsprechenden Typ des Ergebnisses hinter den Namen bzw. hinter die Parameterliste setzen, z.B. `erreichtePunkte(): Ganzzahl` oder `verschluesselt (Zeile: Zeichenkette, Schluessel: Ganzzahl): Zeichenkette`. Das Beispiel zeigt außerdem, dass wir in Blocknamen keine Umlaute verwenden. Das liegt daran, dass Programmiersprachen in der Regel auf der englischen Sprache aufbauen.

### 1.2.3. Prädikat

Ein Prädikat liefert ähnlich wie ein Reporter einen Wert, aber während ein Reporter beliebige Werte oder sogar ganze Listen liefern kann, liefert ein Prädikat nur einen Wahrheitswert, also wahr oder falsch.

Es gibt einen kleinen Trick, wie man Prädikatsblöcke einfacher gestalten kann. Nehmen wir an, ein Spieler hat bei einem Spiel maximal 10 Versuche. Sonst verliert er. Dann könnte man einen Block **gewonnen** wie folgt gestalten:

Abbildung 1.14.: Prädikat

Eleganter geht es, wenn einfach die Bedingung zurückgegeben wird:

Abbildung 1.15.: verbessertes Prädikat

Auch ein Prädikat kann nie allein für sich in einer Programmzeile stehen.

Selbstgeschriebene Prädikate im Quelltext kennzeichnen wir, indem wir einen **Doppelpunkt** und das Wort **Wahrheitswert** hinter den Namen bzw. hinter die Parameterliste setzen, z.B. `gewonnen(): Wahrheitswert`.

### 1.2.4. Hut-Blöcke

Abbildung 1.16.: Beispiele für Hut-Blöcke

Am Anfang der meisten Skripte steht ein Hut-Block, der angibt, wann das Skript ausgeführt werden soll. Namen von Hut-Blöcken beginnen typischerweise mit dem Wort `WHEN`. Ein Skript muss nicht zwingend einen Hut-Block haben, aber wenn nicht, dann wird das Skript nur ausgeführt, wenn der Benutzer auf das Skript selbst klickt. Ein Skript kann nicht mehr als einen Hut-Block haben, und der Hut-Block kann nur am Anfang des Skripts verwendet werden; seine markante Form soll Sie daran erinnern.

### 1.2.5. C-förmige Blöcke

Die meisten Command-Blöcke haben die Puzzleteil-Form, aber einige, wie eine Reihe

Abbildung 1.17.: Beispiele für C-förmige Blöcke

von Wiederholungsblöcken, sind C-förmig. Die meisten C-förmigen Blöcke befinden sich in der Palette „Steuerung". Der Slot innerhalb der C-Form ist eine spezielle Art von Eingabeslot, der ein Skript aus Command-Blöcken als Eingabe akzeptiert und nicht nur einen einzelnen Reporter oder ein Predicate.

C-förmige Blöcke treten vor allem bei Wiederholungsschleifen auf und bei Verzweigungen. Die zweiseitige Verzweigung `IF ... ELSE` hat sogar zwei C-förmige Slots, für jede Seite der Verzweigung einen.

## 1.3. Variable in Snap!

In der Programmierung ist eine **Variable** ein Wert, auf den das Programm zugreifen kann und der sich im Laufe des Programms ändern kann. Anschaulich kann man sich eine Variable vorstellen als eine beschriftete Schublade innerhalb des Speichers. Dabei muss man den Namen und den Inhalt der Variable unterscheiden. Eine Variable `punktzahl` kann z.B. unterschiedliche Werte annehmen. Durch Aufruf des Namens kann ein Programm auf den aktuellen Wert von `punktzahl` zugreifen. Eine Variable ist also die einfachste Möglichkeit, Daten während des Programmablaufs zu speichern.

Snap! kennt verschiedene Arten von Variablen mit unterschiedlicher Reichweite.

### 1.3.1. Globale Variable

Über den Button **Make a variable** in der `Variables`-Palette können neue Variable angelegt werden. Dabei hat der Benutzer die Wahl, ob die Variable **for all sprites** als globale Variable (das ist standardmäßig eingestellt) oder **for this sprite only** nur für das aktuelle Sprite erzeugt werden soll. Die zweite Option ist z.B. dann sinnvoll, wenn sich mehrere gleichartige Sprites auf dem Bildschirm bewegen, z.B: ein Fischschwarm. Dann kann jedes dieser Sprites über eigene Variable **Richtung** und **Geschwindigkeit** verfügen, die die Bewegung steuern, ohne dass Richtung und Geschwindigkeit verschiedener Fische durcheinander geraten.

Globale Variable sollten **sehr sparsam verwendet** werden. Soweit möglich, sollten Parameter oder Skriptvariable eingesetzt werden. Je mehr globale Variable ein Programm hat, desto unübersichtlicher wird es. Es lohnt sich auch, die Namen zwar kurz, aber aussagekräftig zu wählen, um die Lesbarkeit zu erhöhen. Ein Programm mit 20 einbuchstabigen Variablen von a bis t kann der Programmierer nach einigen Wochen selbst nicht mehr lesen.

Auf globale Variable kann man überall in einen Programm zugreifen. Auf Variable, die nur für ein Sprite sichtbar sind, kann auch nur dieses Sprite zugreifen. Sie werden auf dem Bildschirm mit vorangestelltem Namen des Sprites angezeigt. In der Variablenliste tauchen sie aber nur auf, wenn das zugehörige Sprite auch das aktuelle Sprite ist, dessen Skripte angezeigt und bearbeitet werden. Im Kontextmenü stehen die globalen Variablen ganz oben (über dem obersten Strich, falls auch andere Variablenarten vorhanden sind).

### 1.3.2. Parameter

Eine weitere Art von Variablen sind Parameter. Parameter werden eingesetzt, um Blöcke an unterschiedliche Werte anzupassen.

So hat der `Move( )Steps`-Block einen Parameter **Anzahl**. Mit Hilfe dieses Parameters

wird beim Aufruf des Blocks bestimmt, wie viele Schritte das Sprite sich in die gegebene Richtung bewegt. Es ist also weder notwendig, im Block eine Benutzerabfrage einzubauen „Wie viele Schritte soll ich gehen?“ noch die Anzahl mit dem Zufallszahlengenerator auszuwürfeln.

Ein zweites Beispiel: Der `pick random ... to ...`-Block hat die Parameter **untere Grenze** und **obere Grenze**. Beim Aufruf des Blocks legt das Programm beide Grenzen fest.

Beim Start eines Blocks erhalten die Parameter also die Werte, mit denen der Block aufgerufen wurde. Anders als bei Skriptvariablen ist es bei Parametern nicht notwendig, ihnen einen Startwert explizit zuzuweisen (Initialisieren von Variablen). Das geschieht beim Aufruf automatisch. Im Gegenteil, eine Zuweisung würde den beim Aufruf übergebenen Wert überschreiben. Damit wäre der Zweck von Parametern, die Blöcke variabel zu gestalten, verfehlt.

**Parameter dürfen nicht in die Liste der Skriptvariablen aufgenommen werden**. Snap! erzeugt in diesem Fall eine lokale Variable mit dem gleichen Namen. Im Kontextmenü ist dann nur die lokale Variable sichtbar, auf den Wert des Parameters kann nicht zugegriffen werden.

Beim Beenden des Blocks werden die Parameter wieder gelöscht.

Man sollte vermeiden, Parametern den gleichen Namen zu geben wie globalen Variablen. Dann kann man nämlich innerhalb des Blocks nicht mehr auf die globalen Variablen zugreifen. Will man keine neuen Namen erfinden, so setzt man ein kleines p für Parameter vor den Namen.

Bei Parametern lässt sich der Typ einstellen, wenn man in Block-Editor einen Rechtsklick auf den Parameter in der Kopf-Zeile macht und „Input Type“ auswählt. Im Beispiel wird dem Parameter „Spalte“ der Typ Number zugewiesen. An dieser Stelle werden nur noch Zahlen als Eingabe akzeptiert.

In der Kopfzeile werden die unterschiedlichen Typen wie folgt angezeigt: Bei `number` wird ein Doppelkreuz hinter den Variablennamen gesetzt, bei `boolean` ein Fragezeichen, bei `list` drei übereinander stehende Punkte und bei `cShape` ein Lambda. `Any Type` und `text` werden in der Kopfzeile nicht gekennzeichnet.

Abbildung 1.18.: Einstellen des Parametertyps

Abbildung 1.19.: Kopfzeile bei unterschiedlichen Typen

**Achtung: Doppelkreuz, Fragezeichen und Lambda dürfen nicht mit Hilfe der Tastatur hinter den Namen gesetzt werden!** Damit wird keine Zuordnung von Typen durchgeführt.

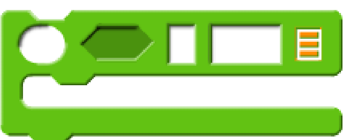


Abbildung 1.20.: Blockdarstellung bei unterschiedlichen Typen

Im Block selbst werden die unterschiedlichen Typen wie folgt dargestellt: `Number` wird durch ein ovales Eingabefeld markiert und `boolean` durch ein sechseckiges. `Any Type` und `text` werden beide durch ein rechteckiges Eingabefeld dargestellt, allerdings ist das Rechteck bei `text` breiter. Drei kleine rote Querstreifen deuten bei `list` eine Liste an. Das `cShape` öffnet eine Klammer zur Aufnahme weiterer Command-Blöcke.

In allen selbstgeschriebenen Blöcken werden die Parameter in runde Klammern gesetzt und hinter einem Doppelpunkt der jeweilige Typ des Parameters angegeben, z.B. `verschluesselt (Zeile: Zeichenkette,`

`Schluessel: Ganzzahl): Zeichenkette`.

Es gibt einen kleinen Trick, wie man das Erstellen von Parametern in selbstgeschriebenen Blöcken vereinfachen kann: Setzt man ein Prozentzeichen vor eine Eingabe, so erkennt Snap! in dieser Eingabe einen Parameter. Gibt man bei der Definition des Blocks im obigen Beispiel ein `verschluesselt( %Zeile , %Schluessel )`, so werden die beiden Parameter bereits eingerichtet und man muss nur noch den Typ durch Rechtsklick einstellen.

### 1.3.3. Skriptvariable

Skriptvariable werden erzeugt, wenn der zugehörige Block ausgeführt wird. Wird der Block beendet, werden die Variablen wieder gelöscht. Skriptvariable werden im Kontextmenü zusammen mit den Parametern in der untersten Abteilung angezeigt.

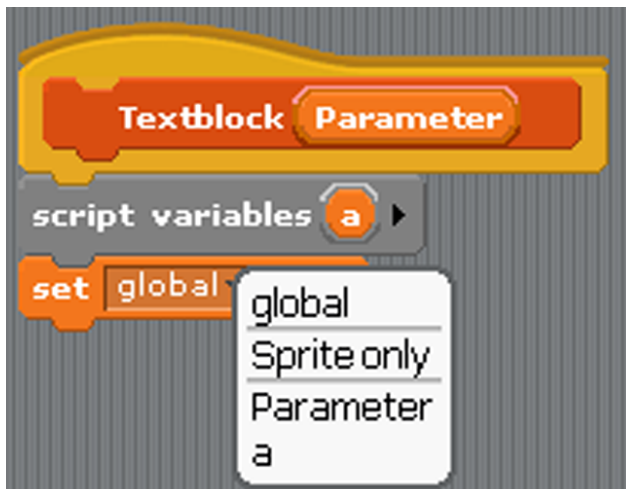



Abbildung 1.21.: Kontextmenü für Variable

Auch bei Skriptvariablen muss man unbedingt vermeiden, sie genau so zu benennen wie globale Variable oder Parameter. Zwar werden dann im Unterschied zu Parametern beide Namen in der Auswahlbox angezeigt, aber man verwechselt leicht die obere und die untere Anzeige und produziert damit schwer einzugrenzende Fehler.

### 1.3.4. Verwendung der Variablenarten

**globale Variable:**

- Globale Variable stehen im gesamten Programm zur Verfügung.
- Globale Variable müssen sparsam verwendet werden.
- Sie sind ggf. initialisieren, z.B. `set myZahl to 1` oder `set myWort to „"`.
- Für globale Variable sind selbsterklärende Namen zu verwenden.

**Parameter:**

- Parameter stehen ausschließlich in dem Block, in dessen Kopfzeile sie definiert wurden, zur Verfügung.
- Parameter dienen zur flexiblen Anpassung von Blöcken.
- Der Typ von Parametern kann vorgegeben werden
- Sie erhalten ihren Startwert beim Aufruf.
- Parameter überschreiben gleichnamige globale Variable.
- Für Parameter sind selbsterklärende Namen zu verwenden.

**Skriptvariable:**

- Skriptvariable stehen ausschließlich in dem Block, im dem sie definiert wurden, zur Verfügung.
- Skriptvarible dienen zur Kapselung von Daten.
- Skriptvariable sind ggf. initialisieren, z.B. `set myZahl to 1` oder `set myWort to „"`.
- Der Typ von Parametern kann vorgegeben werden.
- Sie erhalten ihren Startwert beim Aufruf.
- Skriptvariable überschreiben gleichnamige globale Variable und Parameter.
- Für Skriptvariable sind selbsterklärende Namen zu verwenden. Eine Ausnahme sind Zählvariable. Hierfür sind einbuchstabige Namen wie i, s oder z üblich.

## 1.4. Aufgaben

**Aufgabe 1.1** *Erläutere, wozu man*

- *ein Command,*
- *einen Reporter oder*
- *ein Predicate*

*verwendet.*

**Aufgabe 1.2** *Erläutere, woran man im Quelltext erkennt, ob ein*

- *Command,*
- *Reporter oder*
- *Predicate*

*vorliegt?*

**Aufgabe 1.3** *Erläutere, welche Funktionen eine FOR-Schleife ausführt.*

**Aufgabe 1.4** *Zeichne ein Struktogramm für folgende Operation: Von einer als Parameter übergebenen Zahl soll, wenn sie gerade ist, 5 subtrahiert werden. Wenn sie ungerade ist, soll sie mit 3 multipliziert werden. Das Ergebnis ist auszugeben.*

**Aufgabe 1.5** *Ein Unternehmen verkauft Waren zu einem festen Grundpreis von 220 €. Bestellt ein Kunde mehr als 10 Exemplare, so erhält er einen Rabatt. Dieser beträgt 10 % des Gesamtpreises. Bestellt er mindestens 50 Exemplare, so erhält er einen Rabatt von 20 %. Gesucht ist ein Block mit dem Parameter* `bestellzahl`*, der zu einer Bestellzahl den Gesamtpreis berechnet und ausgibt.*

**Aufgabe 1.6** *Gesucht ist ein Programm, das drei positive Zahlen a, b, c in aufsteigender Größe einliest, die die Längen der Seiten in einem Dreieck bilden. Dann gibt es vier Möglichkeiten:*

1. *die drei Seiten bilden kein Dreieck,*
2. *die drei Seiten bilden ein gleichseitiges Dreieck,*
3. *die drei Seiten bilden ein gleichschenkliges Dreieck,*
4. *die drei Seiten bilden ein ungleichseitiges Dreieck.*

*Das Programm soll die Eingaben und die Art des Dreiecks ausgeben.*

*Die Dreiecksungleichung besagt, dass eine Dreiecksseite höchstens so lang ist wie die Summe der beiden anderen Seiten.*

**Aufgabe 1.7** *Schreibe einen Block mit dem Parameter* `n` *, der die Fakultät einer natürlichen Zahl $fak(n)$ berechnet und ausgibt. Dabei gilt $fak(0) = 1$ und $fak(n) = 1 \cdot 2 \cdot ... \cdot n$, also z.B. $fak(6) = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6$.*

**Aufgabe 1.8** *Zeichne ein Struktogramm für folgende Operation: Für eine als vierstellige Zahl eingegebenes Jahr soll überprüft werden, ob es sich um ein Schaltjahr handelt. Das Ergebnis ist auszugeben.*

*Ein Schaltjahr liegt vor, wenn die Jahreszahl durch 4 teilbar ist. Davon ausgenommen sind die Jahre, die durch 100 teilbar sind, außer denen, die durch 400 teilbar sind.*

**Aufgabe 1.9** *Für einen Test soll 1000 mal mit einem Laplace-Würfel [4] gewürfelt werden. Die Würfe mit einer 6 sollen gezählt werden. Am Ende des Würfelns soll geprüft werden, inwieweit das Ergebnis prozentual vom Erwartungswert abweicht. Das Ergebnis ist auszugeben. Zeichne das zugehörige Struktogramm.*

**Aufgabe 1.10** *Gegeben sind vier Zahlen, die einer Operation als Parameter A, B, C und D übergeben werden zusammen mit einem Parameter Max vom Typ Wahrheitswert. Ist Max true, dann soll die größte Zahl ausgegeben werden. Ist Max false, dann soll die kleinste Zahl ausgegeben werden. Stellen Sie dies als Struktogramm dar.*

[4] Ein Laplace-Würfel ist ein idealer sechsseitiger Würfel, bei dem alle Seite gleich oft vorkommen

# 2. Werkzeuge

## 2.1. Notation von Snap!

Einer der wichtigsten Operatoren im Fach Informatik ist „implementieren“. **„Implementieren“ bedeutet immer, dass ein Verfahren oder eine Datenstruktur in Code, d.h. in Quelltext zu schreiben ist.** Für die handschriftliche Notation von Snap!-Quelltext orientieren wir uns an den Vorgaben der „Ergänzenden Hinweise zum Kerncurriculum Informatik“ in der aktuell letzten Fassung von Juni 2021.[1] Daraus ergeben sich folgende Regeln:

- Wir verwenden ab Klasse 11 einheitlich die **englische Schreibweise** für die Blöcke.

- Der Kopf einer Operation (die „Ergänzenden Hinweise“ verwenden die Bezeichnung „Signatur“) wird wie folgt notiert, um die Bezeichnung, die Übergabeparameter und den Rückgabewert zu kennzeichnen:

  ```
  Bezeichnung(Parameterbezeichnung:
  Parametertyp, ...): Rückgabetyp
  ```

- Variablen, d.h. sowohl Bezeichnungen für Operationen als auch für Parameter, werden in der InFixCap-Schreibweise geschrieben. Der erste Buchstabe wird klein und jeder erste Buchstabe jedes neuen Wortes wird groß geschrieben. Diese Schreibweise wird auch im Abitur verwendet, z.B. `istLeer`. In eigenen Variablenbezeichnungen sind **Leerzeichen und Umlaute** zu **vermeiden**.[2]

[1] Diese Hinweise sollte man unbedingt zur Kenntnis nehmen, wenn man Informatik in der Qualifikationsphase belegen will.

[2] Für die meisten Programmiersprachen sind Leerzeichen Trennzeichen. Sie verstehen auch keine Umlaute. Dazu findet sich Näheres im Kapitel über den ASCII-Code.

- Runde Klammern markieren den Beginn und das Ende von Parameterlisten. Sie werden auch gesetzt, wenn keine Parameter vorhanden sind, z.B. `istLeer()`.

- Aufeinanderfolgende Parameter werden durch Komma getrennt.

- Wenn für Parameter bestimmte Typen definiert werden, dann werden diese im Quelltext mit „Doppelpunkt Typ“ notiert, z.B. `einfuegen(position: Zahl, inhalt: Liste)`

  Hier darf bei `position` nur eine Zahl eingesetzt werden, bei `inhalt` eine Liste. Die Typen werden **nur in der Kopfzeile** angegeben, **nicht im Quelltext!**

- Reporter-Blöcke werden gekennzeichnet, indem wir an der Parameterliste einen Doppelpunkt setzen, gefolgt vom Typ, z.B. `cryptCaesar(text, key): Text`.

- Prädikat-Blöcke werden gekennzeichnet, indem wir an das Ende des Namens bzw. der Parameterliste einen Doppelpunkt setzen, gefolgt von “Wahrheitswert“, z.B. `istLeer(): Wahrheitswert`.

- Textvariable müssen in der Regel initialisiert werden, d.h. die Null, die Snap! automatisch bei der Vereinbarung standardmäßig einsetzt, muss gelöscht werden. Dies schreibt man mit zwei Anführungszeichen: `set MyWord to „"`

- Der Inhalt von Listen wird - wie in der Mathematik - in geschweiften Klammern dargestellt:

  ```
  set Kartenfarbe to list {Kreuz,
  Pik, Herz, Karo}
  ```

- Einrückungen im Snap! werden durch jeweils drei Zeichen Einrückung im Quelltext dargestellt. Gerade in handschriftlichen Texten, die während Klausuren unter Zeitdruck entstehen, ist die Einrückung oft nicht eindeutig lesbar. Spätestens beim Seitenwechsel geht schnell die Übersicht verloren. **Es wird daher dringend empfohlen, in diesem Fall die jeweilige innere Reichweite der Schleife bzw. Verzweigung durch eine Klammer zu kennzeichnen!**

```
sort(myL: Liste): Liste
script variables i, n, hilf
set i to 1
repeat until i > length of myL -1
[  set n to i+1
   repeat until n > length of myL
   [  if item n of myL < item i of myL
      [  set hilf to item n of myL
         replace item n of myL with item i of myL
         replace item i of myL with hilf
      change n by 1
   change i by 1
report myL
```

Abbildung 2.1.: Beispiel für die Klammerung im Quelltext

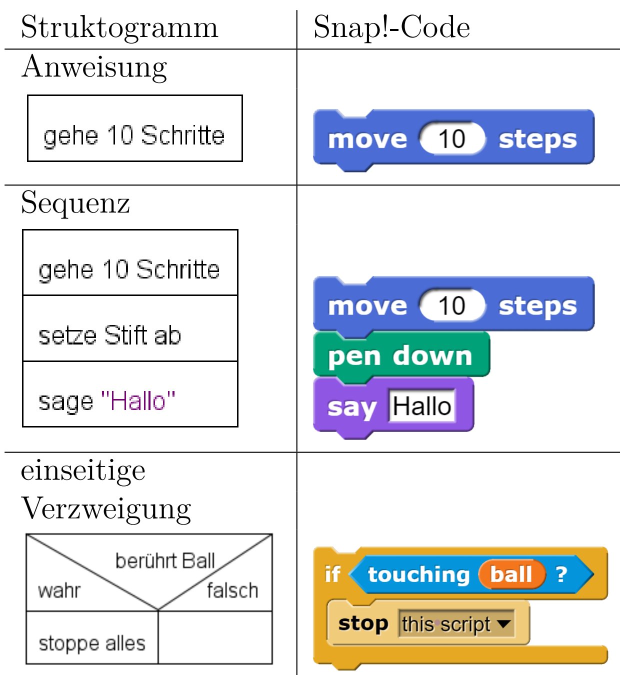
## 2.2. Struktogramm

In den sechziger Jahren entstanden **Programmablaufpläne**, auch Flussdiagramme genannt, als eine Möglichkeit zur grafischen Darstellung von Algorithmen zur Unterstützung der Programmierung. In diesen Programmablaufplänen gab es die Möglichkeit, von einem Punkt über eine Linie an eine andere Stelle im Plan zu springen. Solche Sprungbefehle sind bei der Programmierung in Assembler bzw. Maschinensprache durchaus üblich, gelten aber seit dem Aufsatz von Edgar Dijkstra, „Go To Statement Considered Harmful“, 1968, als schädlich in höheren Programmiersprachen. Außerdem konnte man so Ablaufpläne zeichnen, die nicht mehr in Quelltext umsetzbar waren.

Isaac Nassi und Ben Shneiderman entwarfen deshalb 1972/73 die nach ihnen benannten Nassi-Shneiderman-Diagramme, auch **Struktogramme** genannt, um die strukturierte Programmierung zu unterstützen. Dazu griffen sie auf frühere Arbeiten zurück, die gezeigt hatten, dass jedes sequentielle[3] Programm durch die Kontrollstrukturen Sequenz, Verzweigung und Schleife dargestellt werden kann, also ohne Sprungbefehle. Deshalb gibt es bei Struktogrammen auch nur diese drei Gruppen von Bausteinen.

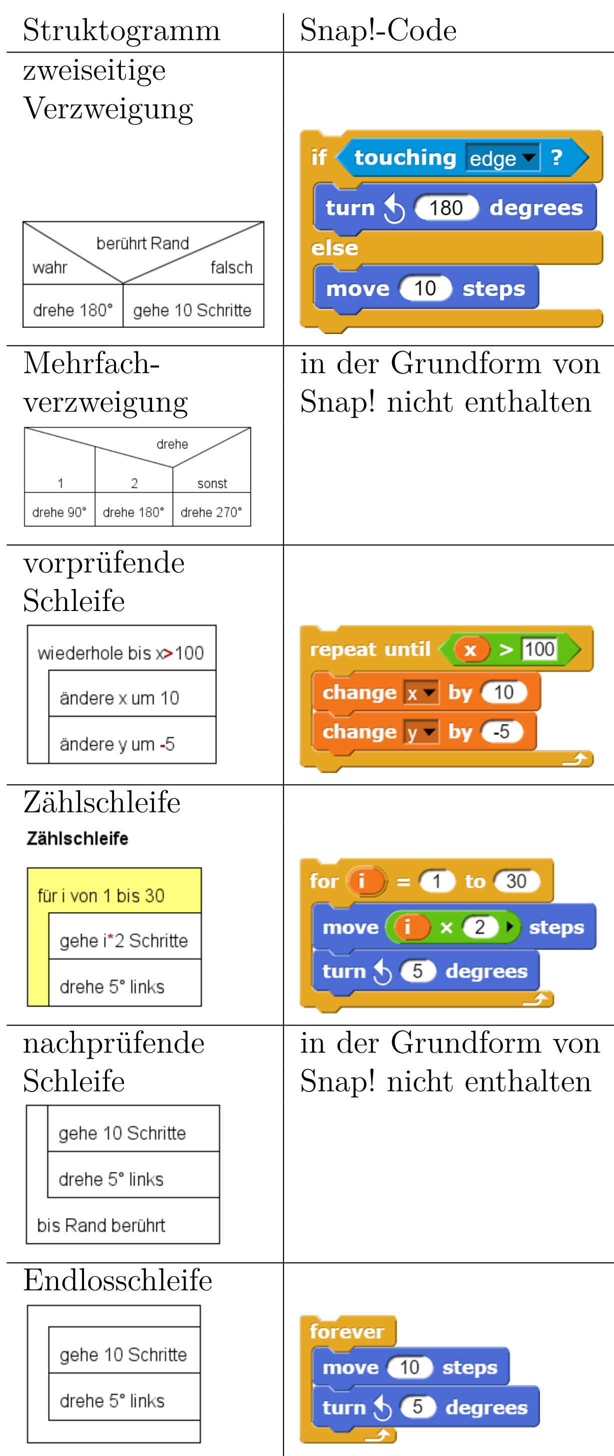
Jedes Struktogramm besteht aus einem **Rechteck**, in das weitere Bausteine eingeschachtelt werden können. Wichtig ist, dass diese Rechteckform im Ganzen auch dann beibehalten wird, wenn durch Verzweigungen die Zahl der Auswahlmöglichkeiten und damit die Anzahl der Spalten wächst. Die vorausgehenden Rechtecke müssen dann breiter gezeichnet werden.

Der Ablauf erfolgt von oben nach unten. Mögliche Bausteine sind:



| Struktogramm | Snap!-Code |
| --- | --- |
| Anweisung: gehe 10 Schritte | move 10 steps |
| Sequenz: gehe 10 Schritte / setze Stift ab / sage "Hallo" | move 10 steps / pen down / say Hallo |
| einseitige Verzweigung: berührt Ball – wahr: stoppe alles; falsch: (leer) | if touching ball ? / stop this script |

[3]Bei sequentiellen Programmen werden die Anweisungen nacheinander ausgeführt. Der Gegensatz dazu wäre parallele Programmierung.



| Struktogramm | Snap!-Code |
|---|---|
| zweiseitige Verzweigung<br>berührt Rand / wahr / falsch<br>drehe 180° / gehe 10 Schritte | if touching edge ?<br>turn ↻ 180 degrees<br>else<br>move 10 steps |
| Mehrfach-verzweigung<br>drehe / 1 / 2 / sonst<br>drehe 90° / drehe 180° / drehe 270° | in der Grundform von Snap! nicht enthalten |
| vorprüfende Schleife<br>wiederhole bis x>100<br>ändere x um 10<br>ändere y um -5 | repeat until x > 100<br>change x by 10<br>change y by -5 |
| Zählschleife<br>**Zählschleife**<br>für i von 1 bis 30<br>gehe i*2 Schritte<br>drehe 5° links | for i = 1 to 30<br>move i × 2 steps<br>turn ↻ 5 degrees |
| nachprüfende Schleife<br>gehe 10 Schritte<br>drehe 5° links<br>bis Rand berührt | in der Grundform von Snap! nicht enthalten |
| Endlosschleife<br>gehe 10 Schritte<br>drehe 5° links | forever<br>move 10 steps<br>turn ↻ 5 degrees |

Nachprüfende Schleifen und Mehrfachauswahl sind in der Grundversion von Snap! nicht enthalten. Sie werden hier aufgeführt, weil sie in Abituraufgaben vorkommen können. Bei Bedarf können nachprüfende Schleifen in Snap! über die Bibliothek (`Libraries`, dann `iteration, composition` wählen nachgeladen werden. Eine Mehrfachauswahl, auch `CASE`-Anweisung genannt, kann durch mehrere geschachtelte Verzweigungen realisiert werden.

Die zweiseitige Verzweigungen ist die Kontrollstruktur, bei der die Darstellung in Snap! am stärksten von der Darstellung im Struktogramm abweicht. Der Kopf einer Verzweigung besteht aus einem Rechteck, in dem zwei Halbdiagonalen die beiden unteren Ecken abtrennen. In dem verbleibenden Dreieck steht die Bedingung. In den beiden Ecken wird wahr/falsch bzw. true/false vermerkt, um die beiden Alternativen zu kennzeichnen. Die beiden alternativen Codeabschnitte stehen **im Struktogramm nebeneinander** statt wie in Snap! untereinander.

Zu jedem Struktogramm gehört eine Kopfzeile mit dem Namen des Struktogramms und der Parameterliste in Klammern, z.B. `zeichneGraph( Knotenliste: Liste)`. Sofern es sich um einen Reporter oder ein Predicate handelt, ist am Ende der Kopfzeile nach einem Doppelpunkt der Typ anzugeben, z.B. `code(data: Zeichenkette, key: Ganzzahl): Zeichenkette` oder `gewonnen(): Wahrheitswert`.

Im ersten Rechteck nach der Kopfzeile werden die Scriptvariablen definiert (in der Fachsprache nennt man das **Deklaration**).

Jede Anweisung erhält ein eigenes Rechteck. Auch mehrere Anweisungen gleicher oder ähnlicher Art dürfen nicht in einem Strukturblock zusammengefasst werden.

Jede Anweisung muss mindestens aus einer Zuweisung bestehen. Eine Zuweisung wird im Struktogramm durch einen nach links gerichteten Pfeil dargestellt. Das Ziel einer Anweisung steht immer links vom Zuweisungszeichen. Rechts davon steht die Quelle, z.B. `Ziel ← Ziel * 3`. **Achtung:** Dieser Linkspfeil darf **nicht** verwendet werden, wenn die Aufgabenstellung lautet „Implementieren Sie eine Operation“. Dann ist grundsätzlich Quelltext verlangt.

Für die Umsetzung des Linkspfeils in Snap!-Quelltext verwendet man den `set to`-Befehl. Auch hier gilt `set ziel to quelle`. Bei Zahlen kann man die Zuweisung auch teilweise auch `change by` ausdrücken: `anzahl ← anzahl + 1` wäre in Snap! `change anzahl by 1`.

Für Struktogramme gilt eine zweite Besonderheit: der Zugriff auf Elemente einer Liste oder anderer Datenstrukturen, bei denen man

auf einzelne Element zugreifen werden kann, wird beschrieben durch den Namen der Datenstruktur, gefolgt von einem Index in eckigen Klammern, z.B. `Liste[i]`. Entsprechend kann man mit zwei Indices auf Elemente einer Tabelle oder Matrix zugreifen, z.B. `tabelle[zeile, spalte]`.

buchstabe ← wort[i]

c ← tabelle[zeile, spalte]

Abbildung 2.2.: Zugriff auf Buchstaben und Listenelemente im Struktogramm

In Snap! muss dieser Zugriff dann mit `letter INDEX of WORD` bzw. `item INDEX of LIST` umgesetzt werden:

set buchstabe to letter i of wort

set c to item spalte of item zeile of tabelle

Abbildung 2.3.: Zugriff auf Buchstaben bzw. Listenelemente in Snap!

Zur Erstellung eines Struktogramms wird das Gesamtproblem, das man lösen will, in immer kleinere Teilprobleme zerlegt, die dann unmittelbar implementiert werden können. Die Fachbezeichnung dafür ist **Top-down-Methode**.

Für Struktogramme gilt die Forderung nach **Allgemeingültigkeit**: Sie sollen keine Befehle enthalten, die speziell aus bestimmten Programmiersprachen stammen. Sie sollen so formuliert werden, dass die zugrunde liegende Logik gut verständlich ist und in jede beliebige[4] Programmiersprache umgesetzt werden kann.

Die Sprache in Struktogrammen grundsätzlich Deutsch, die Sprache bei der Implementierung von Quelltext ist grundsätzlich Englisch. Struktogramme bieten mehr sprachliche Freiheiten in der Formulierung als Quelltext, wo man sich an die vorgegebenen Formulierungen halten muss.

In der professionellen Softwareentwicklung werden Struktogramme heute eher selten eingesetzt, weil inzwischen ausgefeiltere Werkzeuge zur Verfügung stehen.[5] In der Ausbildung von Informatikern sind sie allerdings nach wie vor weit verbreitet, weil sie die Grundbestandteile eines Algorithmus sichtbar machen und dadurch das Verständnis für dessen Funktionsweise befördern. In Klausur- und Abituraufgaben ist der Anteil von Aufgaben, die die Erstellung oder Untersuchung eines Struktogramms erfordern, fast genau so hoch wie der Anteil von Implementierungsaufgaben, die das Schreiben von Quelltext verlangen.

Es gibt eine Reihe von freier Software, die man zum Erstellen von Struktogrammen einsetzen kann. Dabei ist natürlich zu beachten, dass man in Klausuren ein Struktogramm händisch mit Hilfe von Lineal und Geodreieck zeichnen muss. Die Struktogramme in diesen Skripten sind mit dem **Structorizer** gezeichnet. Sie finden die Downloadseite, wenn Sie im Internet nach „Structorizer“suchen. Dabei handelt es sich um ein französisches Produkt. Die deutsche Schreibweise ist in der DIN 66261 genormt. In einigen Kleinigkeiten weicht sie von der internationalen Schreibweise ab. Der Structorizer verfügt im Menu **Diagramme** über einen Schalter **DIN 66261?**, mit dem die deutsche Schreibweise eingestellt werden kann. Man erkennt den Unterschied am einfachsten an der `FOR`-Schleife: die deutsche Version hat unten keinen Querbalken, die internationale schon.

**Achtung, Fehler:** Vorprüfende Schleifen mit einem Querbalken unten bzw. nachprüfende Schleifen mit einem Querbalken oben sind nach DIN 66261 nicht zulässig und deshalb falsch. Nur die Endlosschleife hat einen oberen und einen unteren Querbalken, enthält aber **keine** Bedingungen. Außerdem können Blöcke zwar komplett in Verzweigungen oder Schleifen eingeschachtelt werden, sie dürfen aber nicht überlappen oder überstehen.

[4]genauer gesagt: in jede prozedurale Programmiersprache

[5]z.B. Aktivitätsdiagramme der UML

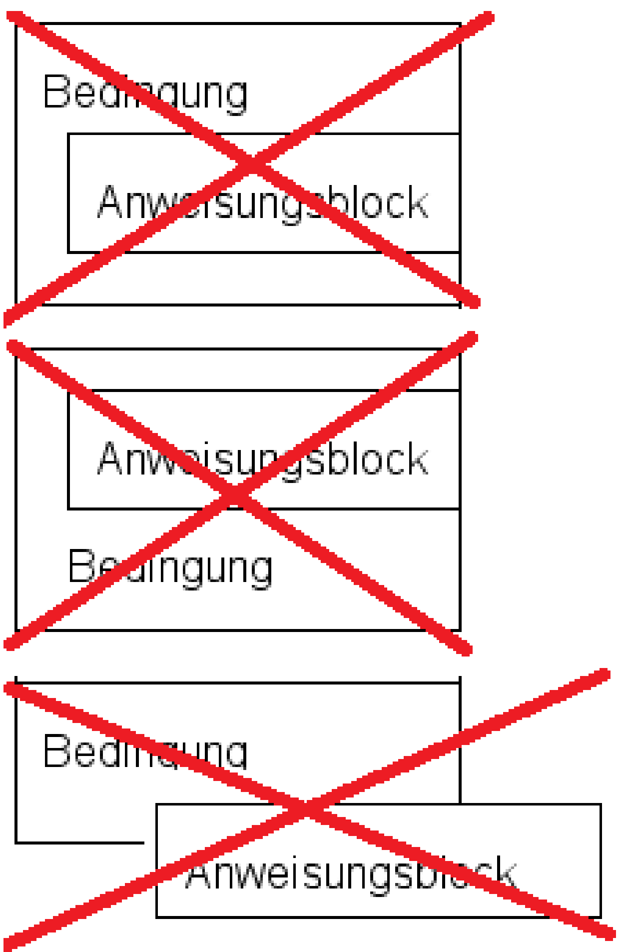



Abbildung 2.4.: Fehler

## 2.3. Tracetabelle

Eine Tracetabelle stellt die Entwicklung der Variablenwerte bei der Abarbeitung eines Algorithmus dar. Tracetabellen werden eingesetzt bei der Fehlersuche in Algorithmen und bei der Untersuchung, was unbekannte Algorithmen leisten.

Wir erläutern das Prinzip einer Tracetabelle am Beispiel der Berechnung des größten gemeinsamen Teilers von a=578 und b=884 und verwenden dazu den im Struktogramm dargestellten Algorithmus.

Der Algorithmus funktioniert, wenn a größer ist als b. Während der Berechnung kann es vorkommen, dass a kleiner wird als b. Dann müssen die beiden Zahlen vertauscht werden. Nun gibt es keinen Befehl, mit Hilfe dessen man den Inhalt von zwei Variablen direkt tauschen könnte. Man kann sich das so vorstellen, dass man in jeder Hand ein dickes Buch hat. Dann kann man die beiden Bücher nicht einfach tauschen. Man muss ein Buch ablegen, kann dann die Hand für das andere wechseln und das abgelegte Buch mit der freien Hand aufnehmen. Genau so verhält es sich mit dem Inhalt zweier Variablen. Man benötigt eine Hilfsvariable. Der Wert der ersten Variable wird in der Hilfsvariable gespeichert. Dann wird die erste Variable mit dem Inhalt der zweiten Variable überschrieben. Zu diesem Zeitpunkt haben beide Variable den gleichen Wert. Dann wird die zweite Variable mit dem Inhalt der Hilfsvariable überschrieben.

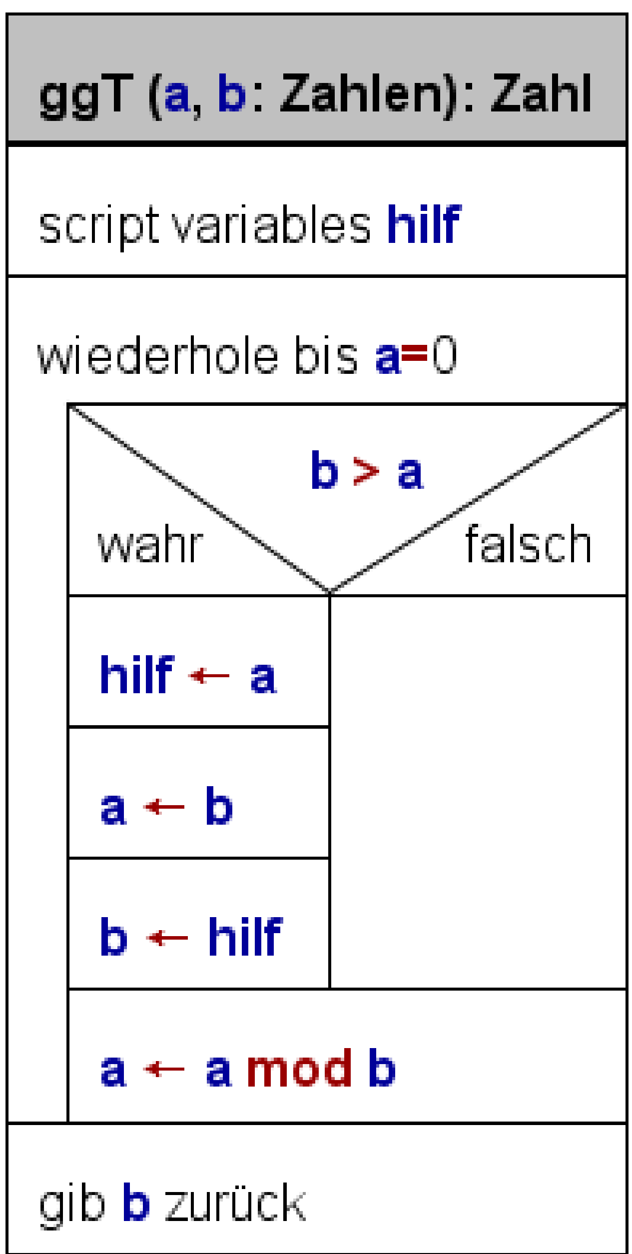



Abbildung 2.5.: Größter gemeinsamer Teiler

Im Struktogramm erkennt man genau diesen Vorgang im `wahr`-Zweig der Verzweigung. `Hilf`

wird mit dem Inhalt von `a` überschrieben, dann wird `a` mit dem Inhalt von `b` überschrieben und schließlich wird `b` mit dem Inhalt von `hilf` überschrieben.

Für alle lokalen Variablen (im Beipiel: `hilf`) und für die Parameter, die sich beim Abarbeiten des Algorithmus verändern (im Beispiel: `a` und `b`), werden Spalten angelegt. Für Parameter, die ihre Eingangswerte beibehalten, braucht man keine eigene Spalte anlegen. Es ist aber auch kein Fehler, wenn man sicherheitshalber ein Spalte für sie anlegt.

| a | b | hilf |
|---|---|---|

In der ersten Zeile geben wir die Startwerte an. Für die Parameter geben wir die Werte an, mit denen die Operation aufgerufen wird. Die lokalen Scriptvariablen werden bei der Erzeugung auf Null gesetzt, weil Snap! so vorgeht.

| a | b | hilf |
|---|---|---|
| 884 | 578 | 0 |

Nun werden die weiteren Zeilen des Algorithmus abgearbeitet. Zunächst wird die Wiederholungsbedingung $a = 0$ geprüft. $a \neq 0$, also wird das Innere der Schleife weiter ausgeführt. An dieser Stelle verändert sich keine Variable. Als nächstes prüfen wir die Verzweigungsbedingung $b > a$. Die ist nicht erfüllt, wir laufen durch den leeren `falsch`-Zweig. Nun gelangen wir zu der Anweisung, wo `a` den Wert `a mod b` erhält. `b` passt ein Mal in `a`, der Rest ist 306. Damit erhalten wir die erste Veränderung eines Werts, `a=306`. Wir tragen den neuen Wert in die nächste Zeile in die `a`-Spalte ein.

| a | b | hilf |
|---|---|---|
| 884 | 578 | 0 |
| 306 | | |

Wir wieder oben bei der Wiederholungsbedingung $a = 9$. Sie ist nicht erfüllt. Die Bedingung der Verzweigung ist jetzt aber erfüllt, $578 > 306$. Wir müssen also den `wahr`-Zweig abarbeiten. In der nächsten Zeile erhält `hilf`

| a | b | hilf |
|---|---|---|
| 884 | 578 | 0 |
| 306 | | |
| | | 306 |

den Wert von `a`. Wir tragen in eine neue Zeile die Zahl 306 in der `hilf`-Spalte ein:

Nun wird `a` mit 578, dem Wert von `b` überschrieben:

| a | b | hilf |
|---|---|---|
| 884 | 578 | 0 |
| 306 | | |
| | | 306 |
| 578 | | |

Schließlich erhält `b` den Wert von `hilf`, nämlich 306.

| a | b | hilf |
|---|---|---|
| 884 | 578 | 0 |
| 306 | | |
| | | 306 |
| 578 | | |
| | 306 | |

Wir beenden den Durchgang, indem wir `a` den Wert von 578 mod 306, nämlich 272 zuweisen.

| a | b | hilf |
|---|---|---|
| 884 | 578 | 0 |
| 306 | | |
| | | 306 |
| 578 | | |
| | 306 | |
| 272 | | |

Die zweite Wiederholung ist beendet, aber die Abbruchbedingung $a = 0$ noch nicht erreicht. In der Verzweigung wählen wir den `wahr`-Zweig. `hilf` erhält den Wert von `a`.

| a | b | hilf |
|---|---|---|
| 884 | 578 | 0 |
| 306 | | |
| | | 306 |
| 578 | | |
| | 306 | |
| 272 | | |
| | | 272 |

`a` wird mit dem Wert von `b` überschrieben:

| a | b | hilf |
|---|---|---|
| 884 | 578 | 0 |
| 306 | | |
| | | 306 |
| 578 | | |
| | 306 | |
| 272 | | |
| | | 272 |
| 306 | | |

Schließlich erhält `b` von Wert von `hilf`:

| a | b | hilf |
|---|---|---|
| 884 | 578 | 0 |
| 306 | | |
| | | 306 |
| 578 | | |
| | 306 | |
| 272 | | |
| | | 272 |
| 306 | | |
| | 272 | |

`a` erhält den Wert von $306 \bmod 272 = 34$:

| a | b | hilf |
|---|---|---|
| 884 | 578 | 0 |
| 306 | | |
| | | 306 |
| 578 | | |
| | 306 | |
| 272 | | |
| | | 272 |
| 306 | | |
| | 272 | |
| 34 | | |

Wir beginnen mit dem dritten Durchgang und dem `wahr`-Zweig. `hilf` erhält den Wert 34:

| a | b | hilf |
|---|---|---|
| 884 | 578 | 0 |
| 306 | | |
| | | 306 |
| 578 | | |
| | 306 | |
| 272 | | |
| | | 272 |
| 306 | | |
| | 272 | |
| 34 | | |
| | | 34 |

`a` erhält den Wert von `b`:

| a | b | hilf |
|---|---|---|
| 884 | 578 | 0 |
| 306 | | |
| | | 306 |
| 578 | | |
| | 306 | |
| 272 | | |
| | | 272 |
| 306 | | |
| | 272 | |
| 34 | | |
| | | 34 |
| 272 | | |

`b` erhält den Wert von `hilf`:

| a | b | hilf |
|---|---|---|
| 884 | 578 | 0 |
| 306 | | |
| | | 306 |
| 578 | | |
| | 306 | |
| 272 | | |
| | | 272 |
| 306 | | |
| | 272 | |
| 34 | | |
| | | 34 |
| 272 | | |
| | 34 | |

Da 272 durch 34 teilbar ist, ist 272 mod 34 Null. Damit erhält `a` den Wert 0.

| a | b | hilf |
|---|---|---|
| 884 | 578 | 0 |
| 306 | | |
| | | 306 |
| 578 | | |
| | 306 | |
| 272 | | |
| | | 272 |
| 306 | | |
| | 272 | |
| 34 | | |
| | | 34 |
| 272 | | |
| | 34 | |
| 0 | | |
| Rückgabewert 34 | | |

Damit die die Abbruchbedingung $a = 0$ erfüllt. Handelt es sich bei dem zu untersuchenden Block um einen Reporter oder ein Predicate, dann ist unter der Tracetabelle der Rückgabewert anzugeben. Wir notieren also den Rückgabewert 34 unter der Tabelle.

Eine Tracetabelle muss nicht vollständig ausgefüllt werden, so dass in jeder Zeile alle Werte eingetragen werden. Es reicht, wenn Einträge in den einzelnen Spalten an den Stellen erfolgen, an denen Veränderungen in den Werten der Variablen auftreten. Vorteilhaft ist auf jeden Fall Kästchenpapier beim händischen Ausfüllen bzw. die Verwendung einer Tabellenkalkulation am PC.

Ein zweites Beispiel zeigt die Verwendung einer Liste, die sortiert werden soll. Die Elemente einer Liste werden notiert, indem wir sie in geschweifte Klammern setzen. Sei `myL = {4, 7, 3, 6}`

```
+sort(+ myL : +)+
script variables i n hilf
set i to 1
repeat until i > (length of myL) - 1
  set n to i + 1
  repeat until n > length of myL
    if item n of myL < item i of myL
      set hilf to item n of myL
      replace item n of myL with item i of myL
      replace item i of myL with hilf
    change n by 1
  change i by 1
report myL
```

Abbildung 2.6.: In-Place-Sortierverfahren

Tabelle 2.1.: Tracetabelle für Sortierverfahren

| myL | i | n | hilf |
|---|---|---|---|
| {4, 7, 3, 6} | 0 | 0 | 0 |
| | 1 | | |
| | | 2 | |
| | | 3 | |
| | | | 3 |
| {4, 7, 4, 6} | | | |
| {3, 7, 4, 6} | | | |
| | | 4 | |
| | | 5 | |
| | 2 | | |
| | | 3 | |
| | | | 4 |
| {3, 7, 7, 6} | | | |
| {3, 4, 7, 6} | | | |
| | | 4 | |
| | | 5 | |
| | 3 | | |

Tabelle 2.2.: Tracetabelle für Sortierverfahren

| myL | i | n | hilf |
|---|---|---|---|
| | | 4 | |
| | | | 6 |
| {3, 4, 7, 7} | | | |
| {3, 4, 6, 7} | | | |
| | | 5 | |
| | 4 | | |
| Rückgabewert {3, 4, 6, 7} | | | |

Das letzte Beispiel befasst sich mit der Bestimmung der Primfaktoren einer gegebenen Zahl n. Jede natürliche Zahl ist entweder eine Primzahl oder sie lässt sich in ein Produkt aus Primfaktoren zerlegen, z.B. $90 = 2 \cdot 3 \cdot 3 \cdot 5$. Wie das Beispiel zeigt, können bei der Primfaktorzerlegung einzelne Primfaktoren auch mehrfach auftreten.

**primfaktoren (n: Ganzzahl): Liste**

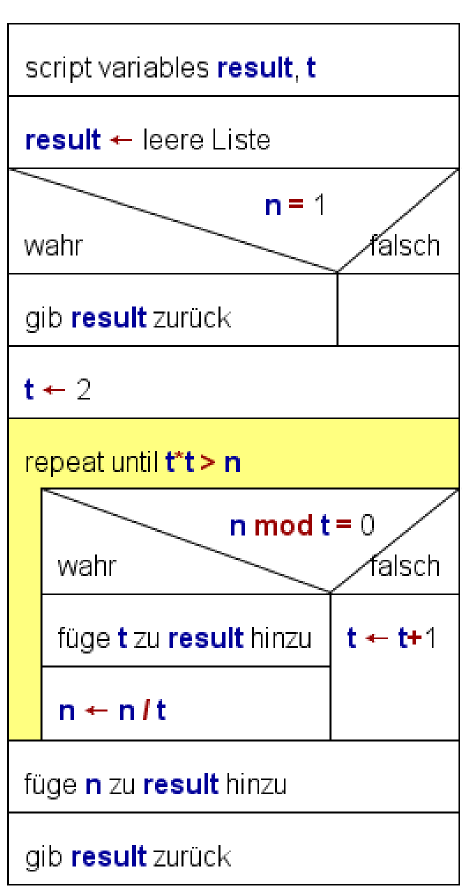



Abbildung 2.7.: Primfaktorzerlegung

Wir behandeln zunächst den Sonderfall: Für $n = 1$ liegt eine Primzahl vor, es gibt also keine Zerlegung, wir geben eine leere Liste zurück. Dann beginnen wir mit dem Teiler $t = 2$ und prüfen, ob n durch t teilbar ist, d.h. ob n mod t = 0. Ist es teilbar, dann fügen wir t zur Liste der Teiler hinzu und setzen n auf $n/t$. Ist es nicht teilbar, dann probieren wir die nächstgrößere Zahl. Der größte Teiler, den wir überprüfen müssen, ist $\sqrt{n}$. Haben wir diese Grenze überschritten, dann haben wir alle Primfaktoren gefunden. Am Schluss müssen wir noch unser aktuelles n in der Liste der Teiler ergänzen, weil es auch ein Teiler des ursprünglichen n ist. Der Algorithmus findet auch Primfaktoren, die mehrfach bei der Primfaktorzerlegung auftauchen, wie das Beispiel mit $n = 126$ zeigt:

Tabelle 2.3.: Tracetabelle zur Primzahlzerlegung von 126

| n | t | result | (n mod t)=0 |
|---|---|---|---|
| 126 | | | |
| | | { } | |
| | 2 | | |
| | | | true |
| | | {2} | |
| 63 | | | |
| | | | false |
| | 3 | | |
| | | | true |
| | | {2,3 } | |
| 21 | | | |
| | | | true |
| | | {2,3,3 } | |
| 7 | | | |
| | | {2,3,3,7 } | |
| Rückgabewert {2, 3, 3, 7} oder $126 = 2 \cdot 3 \cdot 3 \cdot 7$ | | | |

# 3. Grafik

In diesem Kapitel lernen Sie

- wie die Bildschirmkoordinaten in Snap! definiert sind,
- welches die wichtigsten Grafikbefehle sind und
- wie man Farben im RGB-Modell speichern kann.

## 3.1. Koordinaten

Die Bildschirmkoordinaten sind in Snap! so definiert, wie wir es vom kartesischen Koordinatensystem aus dem Mathematikunterricht her gewohnt sind. Der Ursprung (0|0) liegt in der Bildschirmmitte.

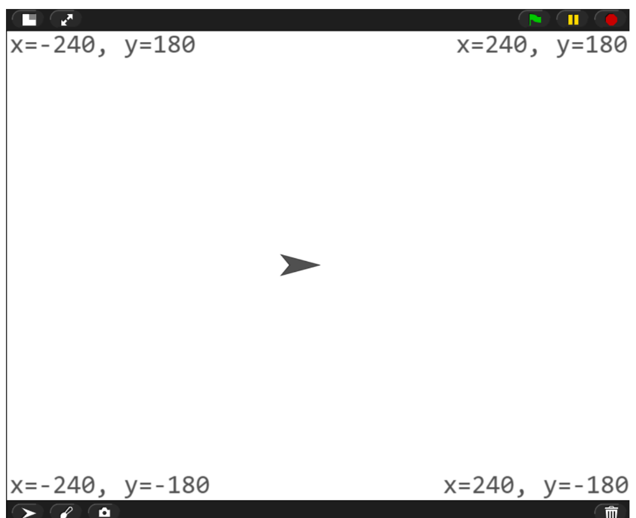



Abbildung 3.1.: Koordinaten der Bildschirmecken

Im Standardfenster stehen 640 Pixel in x-Richtung und 480 Pixel in y-Richtung zur Verfügung. Das bedeutet, dass die kleinste x-Koordinate -320 ist und die größte x-Koordinate +320. Die kleinste y-Koordinate ist -180 und die größte +180. Bei Bedarf lässt sich die Fenstergröße auch auf andere Werte einstellen.

## 3.2. Grafikbefehle

Zum Lernen der Grafikbefehle verwenden wir den Kurs „The Beauty and Joy of Computing", kurz „BJC". Der Kurs ist erreichbar über die Adresse **bjc.edc.org**, wo man „BJC Curriculum" wählen muss. Alternativ kann man auch nach **bjc berkeley** suchen und wählt auf dem obersten Suchergebnis „Try the Curriculum" und dann „BJC Curriculum".

Die Labs 1 bis 3 der Unit 1 Introduction to Programming enthalten zahlreiche Snap!-Befehle. Für unser Langzeitgedächtnis beschränken wir uns auf folgende Auswahl:

1. Aus der **Control-Palette**
   - `repeat anzahl` Wiederholt die in die Klammer eingeschlossenen Anweisungen so oft, wie Anzahl angibt.
   - `if bedingung ...` falls die Bedingung erfüllt ist, werden die Anweisungen in der Klammer ausgeführt, sonst nicht.
   - `if bedingung ... else ...` falls die Bedingung erfüllt ist, werden die Anweisungen in der ersten Klammer ausgeführt, falls nicht, werden die Anweisungen in der zweiten Klammer ausgeführt.
   - `warp` Der Warp-Block stoppt nicht nur alle anderen Skripte, sondern auch das automatische Update der Zeichenoberfläche, das sehr viel Rechnerleistung verbraucht. Erst wenn ein Warp-Block beendet wird, dann werden alle Änderungen, die der Block vorgenommen hat, auf einmal auf der Zeichenfläche sichtbar gemacht.
2. Aus der **Motion-Palette**
   - `go to random position / mouse-pointer / center` Das betreffende Sprite wird an eine

zufällig Position / die Position des Mauszeigers / die Bildschirmmitte gesetzt.

- `go to x: ... y: ...` Das betreffende Sprite wird auf die angegebenen Koordinaten gesetzt. x=0 y=0 ist die Bildschirmmitte.
- `move ... steps` Das betreffende Sprite wird um die angegebene Anzahl von Pixeln in die aktuelle Richtung bewegt.
- `turn right ... degrees` Die Richtung wird um die angegebene Gradzahl nach rechts gedreht.
- `point in direction ...` Falls das betreffende Sprite nicht bereits in die angegebene Richtung zeigt, wird es in diese Richtung gedreht. Dabei sind die Richtungen in Grad angegeben. 0 steht für oben, 90 für rechts, 180 für unten und 270 für links.
- `point towards random position / mouse-pointer / center` Das betreffende Sprite wird in Richtung auf eine zufällig Position / in Richtung des Mauszeigers / der Mitschirmmitte gedreht, wenn es nicht bereits dahin zeigt.

3. Aus der **Pen-Palette**
   - `clear` löscht die Spuren des Zeichenstifts (Pen) und die Stempelspuren (Stamp) vom Bildschirm.
   - `fill` füllt den umrandeten Bereich, in den der Cursor zeigt, mit der aktuellen Farbe.
   - `pen up/down` hebt den Zeichenstift von der Zeichenfläche ab bzw. setzt ihn auf die Zeichenfläche. Nur bei pen down wird gezeichnet.

4. Aus der **Operators-Palette**
   - `pick random ... to ...` liefert eine Zufallszahl zwischen der angegebenen unteren und der angegebenen oberen Grenze. Voreingestellt sind 1 und 10.
   - die Rechenoperationen `+`, `-`, `*`, `/`.
   - die Vergleichsoperatoren <, >, `=`.

5. Aus der **Variables-Palette**
   - `set variable to wert` weist der angegebenen Variablen den rechts angegebenen Wert zu.

Weitere Anwendungsbeispiele für Grafikbefehle finden sich im Kapitel Verschlüsselungsverfahren in Abschnitt über Balkendiagramme und in der zweiten Übungsklausur.

## 3.3. Die kleine Stadt

**Aufgabe 3.1** *Zeichnen Sie durch Klick auf die grüne Flagge eine kleine Stadt auf dem Bildschirm mit Häusern und Zäunen unterschiedlicher Größen.*

Eine mögliche Lösung könnte wie folgt aussehen (Sie können das natürlich viel besser!):

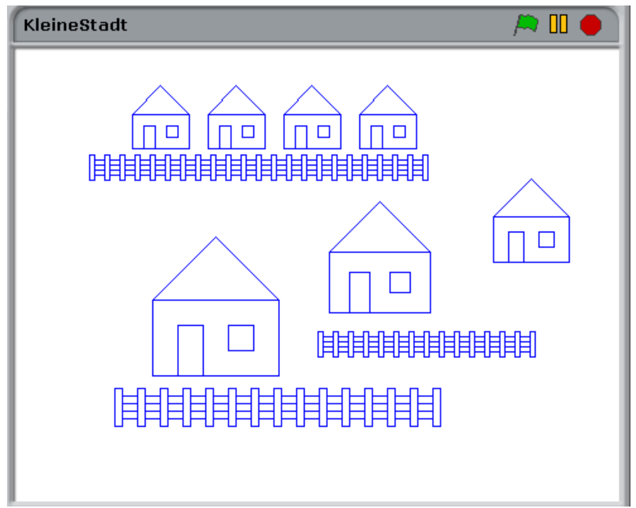



Abbildung 3.2.: Beispiel kleine Stadt

Wichtig ist dabei, dass das Programm über einen Block `Haus` und über einen Block `Zaun` verfügt, deren Größe durch Parameter eingestellt werden kann. Für die Häuser unterschiedlicher Größe sollen nicht verschiedene Blöcke verwendet werden, sondern ein Block mit einem Parameter `breite`. Das gleiche gilt für die Zäune, wobei hier `hoehe` und `pfostenzahl` die Parameter sein sollen. Die beiden Blöcke sollen bei Zeichnen auf Blöcke wie `Rechteck` und -

beim Haus - `Dreieck` zurückgreifen. Natürlich sind auch weitere Ergänzungen möglich, z.B. Hochhäuser aus Rechtecken oder Kugelbäume aus Kreisen und Rechtecken. Die Zeichenbefehle finden wir in der dunkelgrünen Palette `PEN` (Malstift). Die Befehle zum Bewegen des Cursurs stehen in der blauen `MOTION-Palette`.

Zu Beginn des Programms müssen ein paar Dinge erledigt werden:

- Wir zeigen den Cursor an (`show`).
- Wir stellen die gewünschte Grundfarbe ein, im Beispiel oben Blau, es könnte aber auch Schwarz oder Rot sein (`set pen color to ...`).
- Wir stellen die Dicke des Zeichenstifts ein (`set pen size to ...`).
- wir legen die Startrichtung fest.

Um verschiedene Grafik-Blöcke zu einem Bild zusammensetzen zu können, müssen die einzelnen Blöcke am Ende ihrer Ausführung eine bestimmte Richtung einhalten. Diese Richtung wird vom Programmierer zu Beginn festgelegt. Wir legen als Startrichtung fest `point in direction 90`, also nach rechts.

Neben der Richtung, in die der Cursor schaut, ist die Stelle wichtig, an der ein Block beendet wird. Wir vereinbaren: Bei einfachen geometrischen Formen wie Quadrat, Rechteck und Dreieck soll der Block am Ende an der gleiche Stelle stehen wie am Anfang. Bei komplexeren Figuren wie Zaun oder Haus beginnt jeder Block links unten und endet rechts unten. Dann können wir unterschiedliche Zäune oder Häuser einfach nebeneinander zeichnen.

Werden mehrere Zeichenblöcke ausgeführt, so muss sich der Zeichenstift über die Zeichenfläche bewegen, ohne unerwünschte Schleifspuren zu hinterlassen. Jeder Zeichenblock, der etwas malt, sollte also zu Beginn den Stift absenken (`pen down`) und am Ende den Stift anheben (`pen up`).

Ein häufiger Anfängerfehler ist, dass das Programm insgesamt geschrieben wird und dann die Fehler gesucht werde, Zur systematischen Entwicklung eines Programms gehört, dass jeder Block getestet wird, bevor er verendet werden kann. Dazu gehört auch, dass man nicht nur prüft, ob die gewünschte Figur gezeichnet wird, sondern auch, ob man durch geeignete Parameter deren Größe verändern kann.

Zunächst benötigen wir einen Block, um **den Stift auf der Zeichenfläche zu bewegen**, ohne Spuren zu hinterlassen. Dieser Block könnte wie folgt aussehen:

Der Block erhält zwei Parameter: Er zieht `rechts` Schritte nach rechts und `hoch` Schritte nach oben. Falls wir nach links oder unten ziehen wollen, geben wir beim Aufruf einfach negative Zahlen ein. Der Zeichenstift wird in diesem Block nicht gesetzt, weil wir ja gerade keine Schleifspuren hinterlassen wollen. Wichtig ist die Drehung am Ende, damit wir wieder in den Ausgangsrichtung (nach rechts) schauen.

Abbildung 3.3.: Block ziehe(rechts, hoch)

Wir benötigen zusätzlich einen Block, um ein Rechteck beliebiger Breite und Höhe zu zeichnen. Beachten Sie dabei, dass wir den Parameter `hoehe` nicht mit dem Umlaut „ö“, sondern mit „oe“ schreiben. Bei der Bezeichnung von Parametern und Variablen sollte man Umlaute und „ß“ unbedingt vermeiden, weil die meisten Programmiersprachen damit nicht zurecht kommen und Fehler produzieren.[1] Für ein Rechteck zeichnen wir zunächst die Breite, drehen um 90°, zeichnen dann die Höhe und drehen wieder um 90°. Damit haben wir die Hälfte des Rechtecks gezeichnet. Wenn wir diese vier Schritte zwei Mal wiederholen, haben wir ein Rechteck gezeichnet, sind wieder am Startpunkt und zeigen wieder in die Startrichtung. Da wir nicht wissen, ob sich eine Zeichenfigur oder eine Bewegung des Stifts anschließt,

[1] Vgl. dazu mehr über den Hintergrund in Kapitel 7.1 ASCII-Code

heben wir den Zeichenstift am Ende des Blocks wieder an.

```
+rechteck(+ breite # +,+ hoehe # +)+
pen down
repeat 2
    move breite steps
    turn ↺ 90 degrees
    move hoehe steps
    turn ↺ 90 degrees
pen up
```

Abbildung 3.4.: Block rechteck(breite, hoehe)

**Aufgabe 3.2** *Erstelle ein neues Snap!-Programm mit zwei Blöcken zur beliebigen Verschiebung des Zeichenstifts und zum Zeichnen eines Rechtecks beliebiger Breite und Höhe. Zu Beginn sollen die vorbereitenden Maßnahmen (siehe oben) durchgeführt werden. Teste die beiden Blöcke und speichere das Programm.*

Wir wollen mit unseren Blöcken Objekte wie Häuser oder Zäune in beliebiger Größe zu zeichnen. Neben der äußeren Größe, wie z.B. der Höhe eines Zauns oder der Breite eines Hauses, gibt es bestimmte innere Proportionen, die eingehalten werden müssen. Dazu verwenden wir die Einheit „Kästchen" (k). Wir zeichnen die Figuren auf Kästchenpapier, um das Verhältnis der einzelnen Größen zu bestimmen. Zu Beginn eines Block wird dann die Größenangabe, z.B. Höhe oder Breite, in Kästchen umgerechnet, die intern verwendet werden.

Wir zeichnen als nächstes einen **Zaun** aus Rechtecken auf Kästchenpapier, z.B. so:

Jeder Zaunposten ist 5 Kästchen hoch und ein Kästchen breit. Die Querbalken sind im Beispiel ein Kästchen hoch und zwei Kästchen breit. Im Programm geben wir die Höhe die Zaunes und die Anzahl der Posten vor. Dann ist ein Kästchen ein Fünftel der Höhe:

Dieser Block berechnet zunächst aus der Höhe die Lägen eines Kästchens (k). Dann

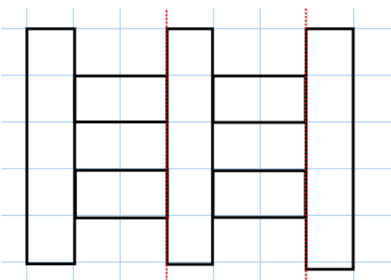


Abbildung 3.5.: Zaun auf Kästchenpapier

```
+zaun(+ hoehe # +,+ postenzahl # +)+
script variables k
set k to (hoehe / 5)
rechteck( k , hoehe )
ziehe( k , k )
rechteck( (2 × k) , k )
ziehe( 0 , (2 × k) )
rechteck( (2 × k) , k )
ziehe( (2 × k) , (-3 × k) )
```

Abbildung 3.6.: 1. Versuch für den `Zaun`-Block

zeichnet er den ersten Teil des Zauns, also einen Pfosten und zwei Querbalken, und bleibt dann unten stehen, wo der zweite Posten beginnt. Man beachte, dass sich der Block nicht mehr um den Zeichenstift kümmern muss. Bei den `rechteck`-Blöcken wird der Zeichenstift automatisch abgesenkt und angehoben. Bei den `ziehe`-Blöcken bleibt der Stift oben.

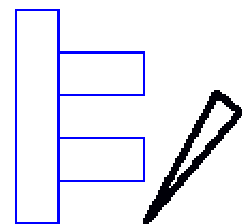


Abbildung 3.7.: Zaunstück

Hat der Zaun sechs Pfosten, so sind fünf Querbalkenpaare notwendig. Bei sieben Posten braucht man sechs Querbalkenpaare usw. Um einen Zaun mit n Pfostenzahl zu zeich-

nen, müssen die drei Rechtecke und die drei Verschiebebefehle (n-1)-mal wiederholt werden und dann noch ein abschließender Pfosten gezeichnet werden:

```
+zaun(+ hoehe # +,+ postenzahl # +)+
script variables k
set k to hoehe / 5
repeat postenzahl − 1
    rechteck( k , hoehe )
    ziehe( k , k )
    rechteck( 2 × k , k )
    ziehe( 0 , 2 × k )
    rechteck( 2 × k , k )
    ziehe( 2 × k , -3 × k )
rechteck( k , hoehe )
ziehe( k , 0 )
```

Abbildung 3.8.: Block zaun(hoehe, pfostenzahl)

**Aufgabe 3.3** *Ergänze das Programm aus der vorigen Aufgabe um einen Block* ***zaun*** *mit den Parametern* ***hoehe*** *und* ***pfostenzahl****. Am Ende soll der Zeichenstift jeweils rechts unten am letzten Pfosten ziehen. Teste den Block mit verschiedenen Höhen und Pfostenzahlen.*

Für ein Hausdach benötigen wir einen Block, der uns ein Dreieck beliebiger Breite zeichnet. Hier gibt es unterschiedliche Möglichkeiten:

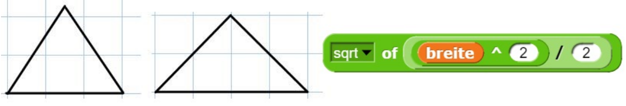



Abbildung 3.9.: Dachformen

Am einfachsten ist das gleichseitige Dreieck (alle drei Seiten sind gleich lang). Es hat den Außenwinkel 120°. Das entspricht einer Dachneigung von 60°. Ein etwas flacheres Dach erhält man mit einem gleichseitigen Dreieck. Die untere Seite hat dann die Länge `breite`, die beiden Schrägseiten haben nach dem Satz des Pythagoras die Länge $\sqrt{(\frac{breite}{2})^2 + (\frac{breite}{2})^2} = \sqrt{\frac{breite^2}{2}}$ (in Snap! oben). Die beiden Außenwinkel an der Dachkante betragen jeweils 135°, die Dachneigung 45°.

**Aufgabe 3.4** *Ergänze in Deinem Programm einen Block* ***dreieck(Breite)****, der Dreiecke unterschiedlicher Breite zeichnen kann, und teste ihn.*

Als nächstes zeichnen wir ein Haus auf Kästchenpapier.

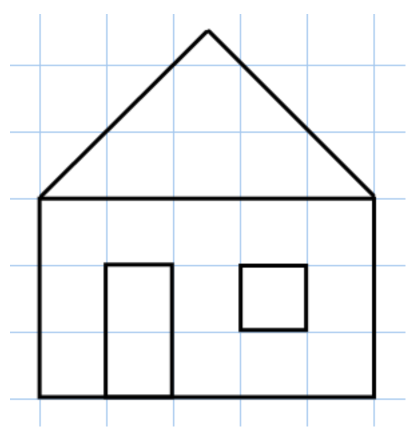


Abbildung 3.10.: Zaunstück

Das Haus kann natürlich auch anders aussehen, z.B. mit je einem Fenster links und rechts von der Tür. Im unserem Beispiel ist das Haus fünf Kästchen breit und 3 Kästchen hoch. Wir benötigen wieder eine Skriptvariable k, wie im `zaun`-Block. Die erste Anweisung wäre in dem Beispiel `set k to breite/5`.

**Aufgabe 3.5** *Ergänze in Deinem Programm einen Block* ***haus(Breite)****, der Häuser unterschiedlicher Breite zeichnen kann, und teste ihn.*

Das Programm kann um Straßen und weitere Gebäude ergänzt werden, z.B. ein Hochhaus mit den Parametern Breite und Stockwerke, eine Kirche, Bäume, usw. Mit dem Befehl `fill` lassen sich die einzelnen Flächen auch farbig ausfüllen.

## 3.4. Galgen zeichnen

In Kapitel 6.6 wird beschrieben, wie man das Spiel Galgenraten implementiert. Eine mögliche Ergänzung ist, dass man in Abhängigkeit von der Variable `fehler` schrittweise den Galgen zeichnet. Dazu müssen wir im ersten Abschnitt des Programms müssen wir den Bildschirm wischen und den Zeichenstift auf die Ausgangsposition und -richtung setzen:

```
go to x: -50 y: -100
point in direction 90
clear
pen down
```

Abbildung 3.11.: Grafik initialisieren

Dann zeichnet man den Galgen mit Hilfe von `Move-` und `Turn-`Befehlen und testet die Lösung. Ist das Ergebnis zufriedenstellend, dann zerlegt man die Sequenz in die einzelnen Zeichenabschnitte. Mit einem Rechtsklick und der Wahl von `ringify` kann man die Abschnitte mit einem grauen Rand versehen. Das bedeutet, dass man jetzt die Befehle selbst in einer Liste speichern kann.

```
move 100 steps
move -50 steps
turn ↺ 90 degrees

move 200 steps
turn ↻ 90 degrees
```

Abbildung 3.12.: Ringify

Wir erstellen eine globale Variable `galgen`, erzeugen daraus eine Liste und fügen den ersten Grafikabschnitt ein. Mit Hilfe des `add`-Blocks können dann die übrigen Abschnitte zu der Liste hinzugefügt werden.

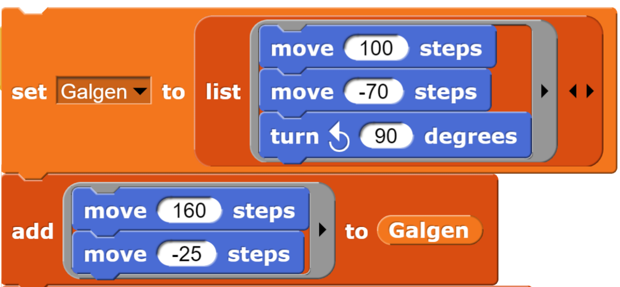



Abbildung 3.13.: Grafikbefehle in einer Liste

Mit dem Befehl `run` im Control-Menue lassen sich die Grafikbefehle in Abhängigkeit von der aktuellen Fehlerzahl dann ausführen:

```
run item Fehler of Galgen
```

Abbildung 3.14.: Run Grafikbefehle

## 3.5. RGB-Farbmodell

Um Farben auf dem Computerbildschirm darstellen zu können, müssen diese codiert werden, d.h. sie müssen als eine Folge von Nullen und Einsen dargestellt werden. Dazu gibt es verschiedene Farbmodelle, die festlegen, wie aus den gespeicherten Daten die dargestellten Farben berechnet werden. Ein weit verbreitetes Farbmodell ist das RGB-Modell. Es wird eingesetzt bei selbstleuchtenden Systemen wie Monitoren. Für das Mischen von Farben auf dem Papier benötigt man andere Farbmodelle.

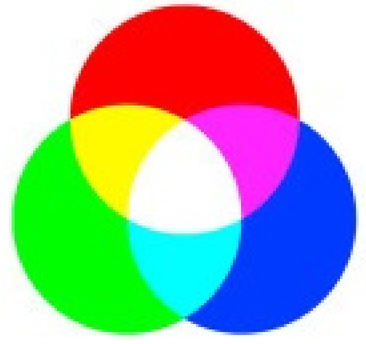


Abbildung 3.15.: RGB-Modell

Seinen Namen trägt das RGB-Modell aus den englischen Bezeichnungen der Grundfarben Red, Green und Blue, aus denen alle anderen Farben zusammengemischt werden. Den

Zusammenhang zwischen der drei Grundfarben und den vier davon abgeleiteten Farben zeigt die Abbildung 4.2.

| **Farbe** | Red | Green | Blue |
|---|---|---|---|
| Rot | 255 | 0 | 0 |
| Grün | 0 | 255 | 0 |
| Blau | 0 | 0 | 255 |
| Gelb | 255 | 255 | 0 |
| Magenta | 255 | 0 | 255 |
| Cyan | 0 | 255 | 255 |
| Weiß | 255 | 255 | 255 |

Tabelle 3.1.: Grundfarben im RGB-Modell

Das RGB-Modell ist ein additives Farbmodell, d.h. je mehr Farbe ein Pixel hat, desto heller ist er. Wenn alle drei Grundfarben (rot, grün und blau) gleichzeitig ihren höchsten Wert annehmen, entsteht aus dem Monitor Weiß. Die unterschiedlichen Anteile für die Grundfarben lassen sich der Tabelle entnehmen, wobei hier 255 der größte Wert ist. Weitere Farben erhält man durch die Wahl anderer Mischungen.

Snap! verwendet das erweiterte RGBA-Modell. Zu den drei Grundfarben tritt noch ein vierter Kanal, der die Transparenz des Bildpunktes angibt. Dabei bedeutet 255 voll deckend, während 0 völlig transparent bedeutet. Mit dem Block `... at ...` aus der **Sensing-Palette** kann man den RGBA-Wert ablesen, wie das Beispiel mit dem Regenbogen zeigt. Der Cursor zeigt auf den roten Streifen des Regenbogens. Der Reporter zeigt, dass der Bildpunkt neben dem vollen Rotanteil auch kleinere Grün- und Blauanteile enthält.

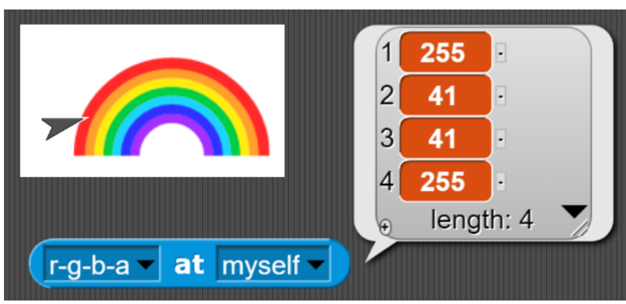



Abbildung 3.16.: RGB-Werte anzeigen

Je mehr Stellen man zur Beschreibung jeder Farbe benutzt, desto feiner werden die Unterschiede und desto mehr Farben kann man darstellen. Bei der in der Tabelle benutzten Variante werden pro Farbe 8 Stellen von Dualziffern benutzt. Damit lassen sich die Zahlen von 0 bis 255 darstellen (vgl. das Kapitel über die Dualzahlen) und somit 256 Abstufungen. Verwendet man 16 Stellen, so sind es schon 65.536 verschiedene Farben und bei 32 Stellen erhält man über vier Milliarden Abstufungen pro Farbe. Das menschliche Auge kann aber nur etwa 500.000 Farbnuancen unterscheiden.

Im diesem Kapitel haben wir gesehen, wie man eine Grafik durch eine Liste von Zahlkombinationen mit jeweils vier Elementen (rot, grün, blau und Alpha) darstellen kann. In den beiden nächsten Kapiteln wollen wir die Blickrichtung umkehren und untersuchen, wie man aus Listen von Zahlkombinationen eine Grafik erstellen kann.

## 3.6. Farbverlauf

Wir beginnen mit einem einfachen Farbverlauf. Wir wollen einen Balken über den Bildschirm zeichnen, der den Farbverlauf von Blau nach Rot zeigt, d.h. der links mit Blau beginnt und bis zum rechten Rand mit Rot endet.

Abbildung 3.17.: Zahlenliste erzeugen

Wir beginnen mit einem Reporter

`zahlenliste`. Der Reporter hat zwei Skriptvariablen `pixels` und `i`. Zunächst wird `pixels` mit Hilfe der Anweisung `set pixels to list{}` zu einer leeren Liste gemacht. Dazu klicken wir auf dem `list`-Block einmal auf den schwarzen Pfeil nach links, um das voreingestellte Element zu löschen. Für unseren Farbverlauf benötigen wir nicht nur einige Hundert, sondern einige Tausend Pixel. Damit wir im weiteren Verlauf nicht ständig warten müssen, fügen wir für die Berechnung der Punkte eine `Warp`-Schleife ein. In den `Warp`-Block kommt eine `FOR`-Schleife, die von 0 bis 255 läuft. Wir fügen mit `add` jeweils den aktuellen Wert von `i` zur `pixels` hinzu und erhalten eine Zahlenliste mit 256 Elementen, die die Zahlen von 0 bis 255 enthält.

Abbildung 3.18.: Farbliste erzeugen

Für das RGBA-Modell benötigen wir aber keine einfache Zahlenliste, sondern eine Liste, deren Elemente aus jeweils vier Zahlen bestehen. Wir ersetzen das `i` im `Add`-Befehl durch eine Liste, die wir durch Klick auf den kleinen Pfeil nach rechts auf vier Elemente erweitern. in das erste Element ziehen wir das `i`. Den Grün-Wert im zweiten Element setzen wir auf Null. Der Blau-Wert an der dritten Stelle soll genau umgekehrt zum Rot-Wert verlaufen. Wir setzen dort einen Minus-Block ein und ergänzen `255 - i`. Den Alpha-Kanal an der vierten Stelle setzen wir auf 255, damit wir die volle Deckung der Farben haben. Wir erhalten jetzt eine Liste von 266 Farben.

Die Farbliste könnte man bereits in ein Kostüm umwandeln und anzeigen lassen. Man würde aber nur einen hauchdünnen Streifen erhalten, den man kaum erkennen kann. Zur

Abbildung 3.19.: Farbstreifen erzeugen

besseren Sichtbarkeit schieben wir um die `FOR`-Schleife noch eine `Repeat 10`-Schleife. Dann wird unser Streifen 10 Pixel dick.

Abbildung 3.20.: Farbstreifen anzeigen

Mit dem Block `new costume` aus der Looks-Palette können wir den Farbstreifen auch anzeigen lassen. `New costume` erwartet drei Parameter: die Farbliste, die uns hier der Block `streifen` liefert, die Breite (hier 256 Pixel) und die Höhe (10, wenn Sie zehn Wiederholungen eingestellt haben).

Abbildung 3.21.: Blau-Rot-Verlauf-Block

Mit drei kleinen weiteren Änderungen erhalten wir den gesuchten Farbverlauf: Wir wollen die ganze Bildschirmbreite ausfüllen und fügen deshalb jeden Punkt zwei Mal mit `Add` hinzu. Dann müssen wir in `new costume` natürlich die Breite auf 512 verdoppeln. Das sind etwas

mehr als die 480 Pixel, die wir in der Breite zur Verfügung haben, schadet aber nicht weiter. Der Block `switch to costume` ermöglicht dann die Darstellung des gesuchten Farbverlaufs.

Abbildung 3.22.: Blau-Rot-Verlauf

## 3.7. Farbige Kreise

Wenn wir den Ball über mehrere Bildschirme fliegen lassen wollen, benötigen wir eine Verbindung zwischen verschiedenen Computern über das Internet. Das leistet ein Dialekt von Snap!, den wir unter **https:\\NetsBlox.org** finden. Wir können auch einfach nach **NetsBlox** suchen. Wir wählen auf der Startseite oben links ▶ **Try Now** und landen unter **editor.netsblox.org/?**im Editor von NetsBlox.

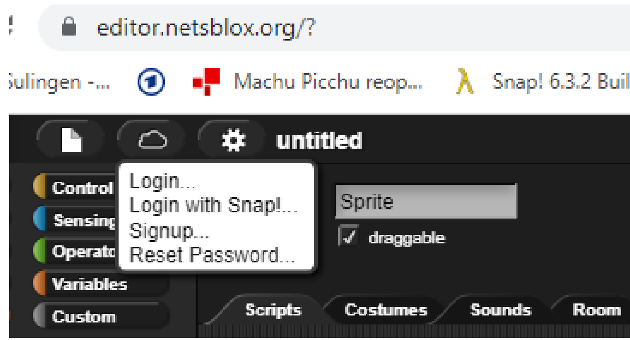



Abbildung 3.23.: Login bei NetsBlox

Auf den ersten Blick sieht er aus wie der Editor von Snap!, nur haben wir zwei Kategorien mehr, nämlich **Network** für die Netzwerkbefehle und und **Custom** für die selbstgeschriebenen Blöcke. Beim genauen Hinsehen erkennt man, dass es im Skriptbereich in der Mitte nicht nur die Reiter für `Scripts, Costumes` und `Sound` gibt, sondern auch einen vierten Reiter für `Room`.

Unter `Cloud` gibt es die bekannten Login-Optionen, aber auch einen zusätzlichen Punkt `Login with Snap!`. Hier funktioniert der Zugang mit dem Benutzernamen und dem Passwort von Snap!, wenn man sich dort bereits angemeldet hat.[2] Man hat zwar die gleichen Zugangsdaten, aber es sind zwei verschiedene Dateiablagen.

**Hinweis:** Leider funktioniert mit der aktuellen Version von **NetsBlox** (v1.33.3) das Kopieren von Blöcken durch Drag-and-Drop des PNG-Bildes noch nicht. Wenn wir also nicht einfach einen farbigen Kreis auf den Bildschirm zeichnen wollen, sondern diesen über mehrere Bildschirme bewegen wollen, dann sollten wir das folgende Skript gleich im Editor von **NetsBlox** schreiben. Laden Sie die übrigen Teilnehmer auch erste dann ein, wenn Sie das Skript fertiggestellt und mit sich selbst getestet haben.

Ein Kreis besteht aus allen Punkten, die von einem gegebenen Punkt den gleichen Abstand haben. Dieser Punkt ist der Mittelpunkt M des Kreises. Der konstante Abstand zum Mittelpunkt ist der Radius r des Kreises. Die Kreisfläche besteht aus allen Punkten, deren Abstand zum Mittelpunkt kleiner als der Radius r ist.

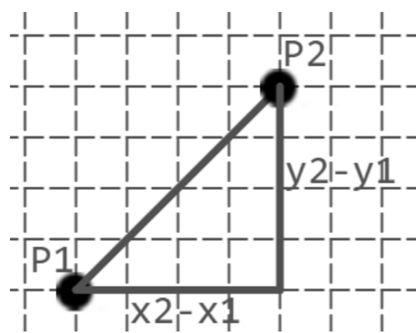



Abbildung 3.24.: Abstand zweier Punkte

Wir benötigen also den Abstand zweier Punkte in einem Koordinatensystem mit x- und y-Achse. Dazu zeichnen wir das Steigungsdreieck ein. Der Abstand ist die Länge der Hypotenuse im Steigungsdreieck. Die Länge der beiden Katheten ergibt sich als Differenz der x- bzw. y-Werte der beiden Punkte. Damit erhalten wir den Abstand mit dem Satz des Pythagoras:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

[2] Ich weiß allerdings nicht, in welchen Abständen die Synchronisierung erfolgt. Es kann sein, dass es bei neuen Snap!-Usern oder wenn man sein Passwort geändert hat, etwas dauert.

Abbildung 3.25.: Abstand zweier Punkte

Wenn wir einen Kreis mit dem Radius $r \in \mathbb{N}$ zeichnen wollen, dann passt dieser Kreis genau in ein Quadrat mit der Seitenlänge $2r$. Wir konstruieren also ein Quadrat mit $2r$ Spalten und $2r$ Zeilen aus $(2r)^2 = 4r^2$ Punkten. Der Mittelpunkt liegt dann bei $(r + 0,5|r + 0,5)$.

Abbildung 3.26.: Kreis

Wir erstellen einen Reporter-Block mit den Parametern Farbe und Radius r, wobei wir für die Farbe eine vierelementige Liste nach dem RGBA-Modell verwenden. Wir sorgen zunächst dafür, dass das Ergebnis eine Liste ist. Damit die Berechnung der Punkte zügig geht (wir müssen ja für jeden Punkt unseren Abstandsblock aufrufen), fügen wir eine `Warp`-Schleife ein. Dann benötigen wir zwei geschachtelte `For`-Schleifen für die Zeilen und die Spalten. Startwert ist jeweils 1, Endwert $2r$. Damit wir den Überblick behalten, nennen wir die Zählvariablen in `z` und `s` um. Innerhalb der Schleifen prüfen wir den Abstand zwischen den Koordinaten des Mittelpunktes und des Punktes. Ist der Abstand kleiner als der Radius, fügen wir die als Parameter übergebene Farbe zum Ergebnis hinzu, falls nicht, fügen wir Weiß (0|0|0) mit der Transparenz 0 hinzu. Am Schluss geben wir das Ergebnis `result` zurück.

Abbildung 3.27.: Kreisskript

Im Hauptprogramm mit den beiden globalen Variablen `r` und `Kreis` stellen wir den Radius ein und erzeugen dann eine Liste für den Kreis. Daraus definieren wir ein neues Kostüm mit der Kantenlänge $2r$.

## 3.8. Der fliegende Ball im Netzwerk

Wir wollen einen Ball auf dem Bildschirm von links nach rechts fliegen lassen, aber nicht nur auf einem Bildschirm, sondern auf mehreren, die im Idealfall nebeneinander stehen.

Abbildung 3.28.: Message-Blöcke

Wir benötigen für unsere fliegenden Bälle zwei Blöcke aus der `Network`-Palette. Wenn wir bei beiden `Message` auswählen können wir bei `msg` die Botschaft eintragen, die jeweils übermittelt werden soll. Unter `send msg` lässt sich hinter `to` auch der Adressat einstellen. Als Botschaft ist auch eine Liste zulässig.

Für den Ball benötigen wir zwei Werte: Die genaue Richtung, in die er fliegt in Grad und den y-Wert, an dem er den linken Bildschirmrand überfliegt.

Das Kreisskript können wir aus dem Kapitel über die farbigen Kreise übernehmen. Leider funktioniert das Kopieren von Blöcken durch Verwendung des PNG-Bildes mit der aktuellen Version von Netblox (v1.33.3) noch nicht Damit wir nicht alles abtippen müssen, klicken

Abbildung 3.29.: Kreisskript

wir mit der rechten Maustaste in Snap! auf das Skript und wählen `script pic ...`. Damit exportieren wir ein PNG-Bild in den Download-Ordner. Wenn wir dieses Bild in NetBlox auf die Skriptfläche ziehen, wird das Skript zusammen mit den verwendeten Blöcken (in diesem Fall `Kreis` und der Abstandsblock `d`) in das neue Programm importiert.

Abbildung 3.30.: Los geht's

Unter `losgehts` fassen wir die Anweisungen zusammen, die wir bei Start des Programms nach dem Erstellen des Balls durchführen. Wir verstecken zunächst den Ball, setzen ihn an die Mitte des linken Bildrandes und zeigen ihn wieder. Die Startrichtung wählen wir zufällig zwischen 70 und 110, also grob nach rechts. Dann kommt die eigentliche Bewegung, die wir so lange wiederholen, bis die x-Position des Sprites größer ist als 220, also kurz vor dem rechten Bildrand. Den `of`-Block finden wir in der `Sensing`-Palette. Standardmäßig heißt er `costume # of ...`. Wir müssen zunächst hinten `Sprite` einstellen und können dann vor `x position` wählen. Aber zurück zur Bewegung:

Wir gehen 5 Schritte in die angezeigte Richtung und warten dann 0,03 Sekunden. Mit den beiden Werten kann man etwas spielen, um das Optimum für sich selbst herauszufinden. Falls wir an den Rand kommen (das kann dann nur noch oben und unten sein), lassen wir das Sprite abprallen.

Soweit die Wiederholung. An der Grenze $x = 220$ angekommen, verstecken wir den Ball wieder und senden dann mit dem `send msg`-Block aus der Netzwerkpalette eine Nachricht an den zweiten Spieler. Als ersten Parameter wählen wir mit dem kleinen schwarzen Pfeil `message` aus, in das dann aufklappende freie weiße Feld setzen wir eine Liste mit zwei Parametern, der y-Position des Sprites und der Richtung (`direction` des Sprites. In späteren Ausbaustufen können wir auch die Geschwindigkeit zu einer Variable machen (bei der Erstellung aber die Option `this sprite only` wählen) und hier als dritten Wert ergänzen. Adressat unserer Nachricht ist Spieler B.

Abbildung 3.31.: Message

Der `when I receive message`-Block ist ganz ähnlich aufgebaut. Wir verstecken das Sprite, gehen an die angegebene Position am linken Bildschirm und stellen die angegebene Richtung ein. Der Mittelteil mit der Bewegung verläuft genau so wie beim `losgehts`-Block. Nur den Adressaten müssen wir anpassen. Bei zwei Spielern ist es der Startspieler A, bei drei oder mehr der jeweils nächste Spieler in der Reihenfolge.

Nun müssen wir uns um die Mitspieler kümmern. Unter `Room` erhalten wir folgendes Bild (natürlich mit Ihrem Benutzernamen und dem Projektnamen, unter dem Sie das Programm gespeichert haben). Wenn Sie das Bild mit Ihrem Bildschirm vergleichen, sehen Sie, dass ich die leere schwarze Fläche zusammengeschrumpft habe:

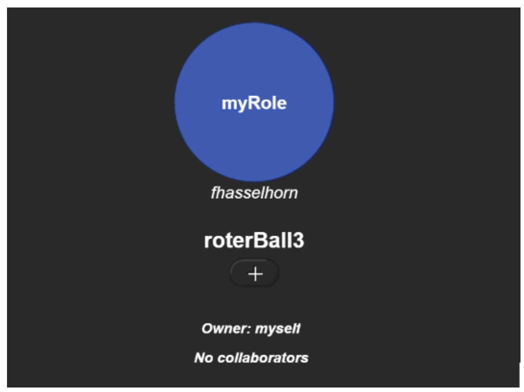



Abbildung 3.32.: Room ohne Mitspieler

Durch Klicken auf das Plus-Zeichen könnten wir weitere Rollen hinzufügen, bis wir die gewünschte Teilnehmeranzahl erreicht haben. Allerdings hätten diese Mitspieler alle leere Skriptflächen, d.h. jeder müsste das Spiel komplett neu programmieren. Ob das dann wirklich zusammenpasst, ist die Frage. Wir wählen deshalb einen anderen Weg und klicken in den farbigen Rollen-Kreis (nicht genau auf den Namen!), Damit erhalten wir ein Auswahlmenü, bei dem wir uns für `Duplicate` entscheiden:

Abbildung 3.33.: erstes Edit-Rolle-Menü

Damit verändert sich die Darstellung unseres Raumes:

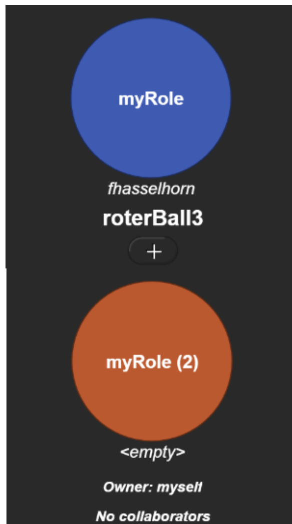



Abbildung 3.34.: Room mit einem Mitspieler

Die neue Rolle hat jetzt alle Skripte, die wir für das ursprüngliche Programm erstellt haben.

Wenn wir auf die neue Rolle klicken, erscheint das Edit-Feld mit zwei zusätzlichen Optionen: Mit `Move To` können wir zu dieser Rolle wechseln, mit `Delete role` die Rolle wieder löschen

Abbildung 3.35.: erweitertes Edit-Rolle-Menü

Der Einfachheit halber verwenden wir hier für die Rollennamen fortlaufende Buchstaben. `MyRole` benennen wir um in `A`, indem Sie in dem farbigen Kreis auf den Rollennamen klicken.

Abbildung 3.36.: Rollen-Namen vergeben

Bevor Sie Multiplayer-Spiele spielen können, müssen alle Spieler vorher angemeldet und eingeloggt sein. Wenn Sie der erste sind, der das Spiel eröffnet (Owner), um Ihre Mitspieler einzuladen, gehen Sie auf die Registerkarte `Room`, wo Sie eine Übersicht über den aktuellen Raum und die verfügbaren Rollen sehen, die wir vorher in ausreichender Anzahl dupliziert haben. Jeder Spieler nimmt eine Rolle ein. Um Spieler einzuladen, klicke auf eine Rolle, wähle `invite user` und suche nach dem Benutzernamen deines Freundes und drücke ok, um ihn einzuladen.

Wenn Sie zu einem Spiel eingeladen werden, vergewissern Sie sich, dass Sie **editor.netsblox.org** in Ihrem Browser geöffnet haben und dass Sie angemeldet sind. Nachdem Sie eingeladen wurden, erscheint ein Dialog, in dem Sie gefragt werden, ob Sie dem Spiel beitreten wollen.

Jeder Spieler muss das Skript `when I receive message` am Ende noch so anpassen,

dass er seine Nachricht an den benachbarten Spieler sendet, dessen Bildschirm als nächster durchflogen wird.

Vor dem Start des Spiels vergewissern Sie sich, dass alle Spieler anwesend sind, indem Sie sich die Raumansicht ansehen. Nun, da Sie alle Vorbereitungen getroffen haben, kann die Hauptrolle das Spiel starten, indem Sie auf die grüne Flagge in der oberen rechten Ecke klicken. Sie können die Bühne (Spielplatz) maximieren, indem Sie auf das Symbol klicken.

Um Multiplayer-Spiele zu testen, können Sie sich selbst zum Spielen einladen, indem Sie eine neue Browser-Registerkarte öffnen, zum NetsBlox-Editor gehen, sich dort anmelden und `myself` als Spieler einladen.

## 3.9. Regeln für Zeichenblöcke

Für das Erstellen von Blöcken für Zeichenprogramme gibt es ein paar Regeln:

- Ähnliche Figuren werden mit Hilfe von Blöcken gezeichnet, die durch Parameter variabel gestaltet werden können. So braucht man nicht drei verschiedene Blöcke für Fenster, Tür und Haus. Vielmehr reicht ein Block `rechteck`.
- In der Hierarchie der Zeichenblöcke werden die Befehle **`pen down`** und **`pen up`** möglichst weit unten eingebaut, also bei den Blöcken, die tatsächlich etwas zeichnen, und nicht bei den Blöcken, die nur Grundfiguren zusammenstellen. Dies macht die Programme übersichtlicher.
- **`go to`**-Befehle mit festen Koordinaten werden **nicht** in Blöcken verwendet. Das führt nur dazu, dass man für jedes Zeichenobjekt dieser Art einen eigenen Block benötigt, was den Programmtext aufbläht. Man kann allerdings `go to`-Befehle mit einer Variable bzw. einer Formel verwenden. Dazu findet sich ein Beispiel bei den Säulendiagrammen in Kapitel 8.5.
- **`point`**-Befehle werden ebenfalls **nicht** in Zeichenblöcken verwendet. Sie führen dazu, dass man eine Figur nicht mehr beliebig drehen kann.
- Jeder einzelne Zeichenblock ist deshalb so zu gestalten, dass der Cursor am Ende in die gleiche Richtung zeigt wie am Anfang.
- Zeichenblöcke sind entweder so zu konstruieren, dass der Cursor am Ende an der gleichen Stelle steht wie am Anfang oder an der Stelle, wo die nächste Figur angesetzt werden soll.
- Der **`clear`**-Block gehört **nicht** in die elementaren Zeichenblöcke, sondern in das Hauptprogramm. Eine Ausnahme wären animierte Figuren.
- Ein Tipp zum Vorgehen: Beginnen Sie mit den einfachen Blöcken und testen Sie jeden dieser Blöcke mit mehreren Werten für die Parameter, bevor Sie sie zusammensetzen.

## 3.10. Aufgaben

**Aufgabe 3.6** *Implementieren Sie ein Programm, das bunte Rechtecke, Dreieck und Kreise auf den Bildschirm zeichnet.*

**Aufgabe 3.7** *Implementieren Sie ein Programm, das einen Galgen aus Strichen auf den Bildschirm zeichnet, wie man ihn beim Galgenraten benutzt.*

**Aufgabe 3.8** *Ergänzen Sie das Programm **Fliegende Bälle***

- *durch verschiedene Bälle, die gleichzeitig fliegen*
- *mit unterschiedlichen Geschwindigkeiten*
- *indem Sie bei Berührung des oberen bzw. unteren Randes die Geschwindigkeit erhöhen*
- *...*

# 4. Mathematische Algorithmen

In diesem Kapitel lernen Sie

- welche Operationen für Variablen vom Typ Zahl geeignet sind,
- wie man mit dem `MOD`-Befehl zyklisches Zählen umsetzt,
- wie man die Belegung von Variablen bei der Ausführung eines Algorithmus in Form einer Tracetabelle darstellt.

## 4.1. Datentyp Zahlen

Snap! kennt nur den Typ Zahl (number). Es wird nicht zwischen Ganzzahlen (integer) und Kommazahlen (real) unterschieden. Angezeigt werden in der Regel neun Nachkommastellen. Snap! gibt als Länge einer periodischen Dezimalzahl aber 17 oder 18 Stellen an. Wie Snap! rechnet, behandeln wir im nächsten Kapitel in den Abschnitten über Wechselgeldautomaten und Sägezahnfunktion. Falls - wie häufig in der Informatik - nur ganze Zahlen zugelassen sind, muss der Programmierer dies sicherstellen, vor allem bei der Division (dazu mehr in 5.2 Rund um `MOD`).

Eine Zahlvariable kann mit zwei verschiedenen Blöcken verändert werden:

Abbildung 4.1.: Set und Change

Der `set`-Befehl ist wie folgt zu lesen: Auf `set` folgt die Variable, also das Ziel, dann `to` und dann der Wert. Eine Variable ist nichts anders als eine Speicherstelle. Durch den `set`-Befehl wird also die entsprechende Speicherstelle mit dem angegebenen Wert überschrieben. Etwa vorher dort enthaltene Werte gehen verloren. Eine kleine Merkhilfe: Der Platz der Variablen ist immer durch das kleine nach unten gerichtete schwarze Dreieck gekennzeichnet. Daraus folgt auch, dass als erster Parameter nicht `item i of Liste` eingesetzt werden kann. Listenelement können nur mit dem `replace`-Befehl geändert werden!

Der `change`-Befehl nimmt die Variable hinter dem `change` und verändert sie um den hinter `by` angegebenen Wert. `Change myVar by 1` ist etwas kürzer als das gleichbedeutende `set MyVar to MyVar+1`, vor allem wenn die Variablennamen länger sind. Der `change`-Block darf **nur mit Zahlen** verwendet werden, sonst entstehen Fehler!

Wird eine neue Variable vereinbart, so erhält sie automatisch den Wert Null. Das gilt nicht nur für Zahlvariablen, sondern für alle Variablen.

Einige wichtige Blöcke für Zahlen:

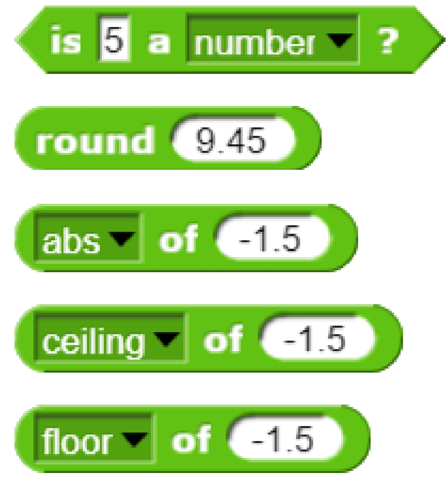



Abbildung 4.2.: Blöcke für Zahlen

`is MyZahl a number?` prüft, ob MyZahl eine Zahl (oder ein anderer Variablentyp) ist.

`abs of MyZahl` liefert den Betrag einer Zahl.

`round MyZahl` rundet eine Zahl nach den mathematischen Regeln, d.h aufrunden ab 0,5,

sonst abrunden.

`ceiling of MyZahl` rundet eine Zahl zur nächst größeren ganzen Zahl auf.

`floor of MyZahl` rundet eine Zahl zur nächst kleineren ganzen Zahl ab.

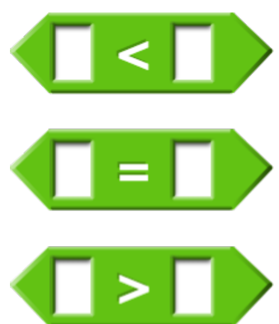



Abbildung 4.3.: Gleichheitsoperatoren

Wichtig sind für den Umgang mit Zahlen die Gleichheitsoperatoren. Hier gibt es zwei Besonderheiten. Stehen auf beiden Seiten des Gleichheitsoperators Buchstaben oder Zeichenketten, dann unterscheidet der Operator nicht zwischen Groß- und Kleinschreibung. `ÄSOP` ist also gleich `äsop`. Außerdem sind über die Auswahl von **relabel** eine Reihe versteckter Operatoren erreichbar, z.B. $\leq$, $\geq$ und $\neq$.

Häufig benötigt man Dezimalzahlen mit einer festen Anzahl von Nachkommastellen, z.B. bei Geldbeträgen. Dazu multipliziert man mit der gewünschten Zehnerpotenz, rundet und teilt dann wieder durch die Zehnerpotenz. Das folgende Beispiel liefert Zahlen mit zwei Nachkommastellen:

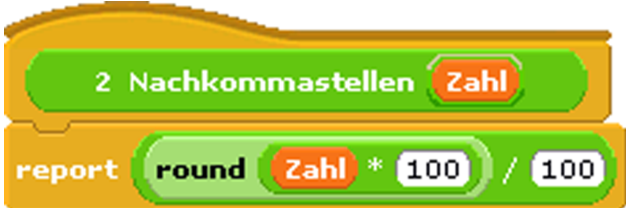



Abbildung 4.4.: Zwei Nachkommastellen

## 4.2. Rund um MOD

Ein wichtiger Befehl bei der Programmierung ist der `MOD`-Befehl. `a mod b` liefert den ganzzahligen Rest bei der Division von a durch b. Beispiel: 13465 mod 100 = 65. Als Reste können alle natürlichen Zahlen auftreten, die kleiner sind als der Divisor, in unserem Beispiel alle Zahlen von 0 bis 99.

### 4.2.1. Zählen mit MOD

In der Informatik müssen wir häufig zählen, aber nach einer bestimmten Anzahl von Schritten wieder bei Eins oder Null anfangen.

a) Mitzählen von 1 bis k

Eine klassische Aufgabe ist, einen Text oder eine Liste beliebiger Länge in eine Tabelle umzusetzen. Dabei läuft eine Zählvariable `i` von 1 bis n, wobei n die Länge des Textes oder der Liste ist. Von dieser Zählvariable abgeleitet ist eine zweite für die Spaltenbreite, die aber nur von 1 bis k ($k < n$) läuft.

Es liegt nahe, hierfür den `MOD`-Befehl zu verwenden. Betrachten wir die Tabelle. In der linken Spalte finden wir die Zählvariable i, die bis zum Ende der Datenstruktur läuft. Sei beispielsweise k=5 (k kann natürlich auch jeden anderen Wert haben). In der zweiten Spalte finden wir dann `i mod 5`. Wir sehen, i mod 5 läuft anfangs parallel zu i. Die erste Abweichung finden wir bei i=5. i mod 5 läuft nicht von 1 bis 5, sondern von 0 bis 4, weil bei der Division durch 5 die Reste von 0 bis 4 auftreten.

| i | i mod 5 | i mod 5 +1 | (i-1) mod 5 +1 |
|---|---|---|---|
| 1 | 1 | 2 | 1 |
| 2 | 2 | 3 | 2 |
| 3 | 3 | 4 | 3 |
| 4 | 4 | 5 | 4 |
| 5 | 0 | 1 | 5 |
| 6 | 1 | 2 | 1 |
| 7 | 2 | 3 | 2 |
| 8 | 3 | 4 | 3 |

Tabelle 4.1.: Tabelle zum Zählen (k=5)

Wir addieren deshalb Eins und erhalten `(i mod 5)+1` in der dritten Spalte. Nun erhalten wir zwar die Zahlen von 1 bis 5, aber nicht an den richtigen Stellen. Statt mit 1 beginnt die Zählung nun mit 2, weil 1 mod 5=1 und 1+1=2. Wir müssen also jeweils Eins abziehen, bevor

wir die MOD-Operation durchführen. Somit erhalten wir in der vierten Spalte `((n-1) mod 5) +1` und diese liefert die gewünschte Zählung von 1 bis 5, beginnend mit 1.

Abbildung 4.5.: Mitzählen von 1 bis k

b) Wiederholtes Zählen von 1 bis n

Beim **wiederholten Zählen** verwenden wir eine Zählvariable i, die nach einem bestimmten Wert n wieder bei Eins anfangen soll. Dazu können wir nicht die Zählvariable einer `FOR`-Schleife verwenden, sondern müssen selbst eine Zählvariable initalisieren (ihr einen Startwert zuweisen, hier 1) und sie inkrementieren (den Wert erhöhen bzw. wieder bei 1 beginnen lassen).

Der Befehl zum Inkrementieren steht jeweils am Ende der Wiederholungsschleife. Mit `set i to i+1` können wir eine Zählvariable i jeweils um 1 weiter zählen. Wenn wir nach n Schritten aber wieder bei Eins anfangen müssen, benutzen wir den MOD-Befehl. `set i to i mod n` liefert `i` für alle Zahlen, die kleiner sind als n. Das kann man in der Tabelle oben unter a) in der zweiten Spalte sehen. Allerdings erhalten wir für $i = n$ den Wert Null. Wir sollen aber wieder mit Eins beginnen neu zu zählen. Also müssen wir rechnen `(i mod n) +1`. Das ergibt die dritte Spalte in der Tabelle, die beim Mitzählen nicht die richtigen Werte liefert. Für das Inkrementieren am Ende einer Wiederholung passt dieser Wert aber genau: Am Ende der ersten Wiederholung wird die Zählvariable auf 2 gesetzt, am Ende der zweiten Wiederholung auf 3, usw. und am Ende der n-ten Wiederholung wieder auf 1.

Abbildung 4.6.: Wiederholtes Zählen von 1 bis n

c) Wiederholtes Zählen von 0 bis n-1

Das Zählen von 0 bis n-1 funktioniert ganz ähnlich wie das Zählen von 1 bis n, nur das wir hier erst Eins addieren und dann den Rest bei der Division durch n bilden:

Abbildung 4.7.: Zählen von 0 bis n-1

d) gerade und ungerade Zahlen

Ob eine Zahl gerade oder ungerade ist, kann man einfach mit dem MOD-Befehl überprüfen. Ist `zahl mod 2` Eins, so liegt eine ungerade Zahl vor. Ist `zahl mod 2` Null, so handelt es sich um eine gerade Zahl.

### 4.2.2. DIV - die ganzzahlige Division

Die Division von natürlichen Zahlen ist die erste Grundrechenart, die nur in einigen Fällen wieder zu ganzzahligen Ergebnissen führt. Deshalb lernen wir in der Grundschule erst die Division mit Rest. Später wird der Zahlbegriff um die Bruchzahlen erweitert. In der Informatik greifen wir aber häufig zurück auf die Division mit Rest. Mit dem `MOD`-Befehl haben wir die Möglichkeit, den Rest zu berechnen. Wenn wir nun zuerst den Rest berechnen, diesen vom Dividenden abziehen und dann teilen, erhalten wir ein ganzzahliges Ergebnis:

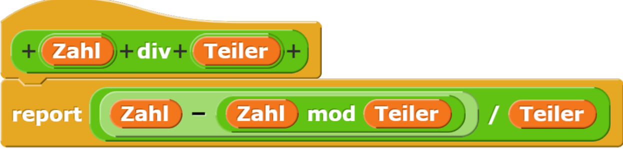



Abbildung 4.8.: DIV

Alternativ können wir auch den Befehl `floor` zum Abrunden auf die nächstkleinere ganze Zahl verwenden: `floor(zahl\teiler)`.

### 4.2.3. Paritätsbit

Ein Paritätsbit ist eine einfache Möglichkeit, zu prüfen, ob eine Dualzahl aus Einsen und Nullen richtig übermittelt wurde. An die Zahl wird eine Prüfziffer angefügt. Beim geraden Paritätsbit wird die Prüfziffer so gewählt, dass eine gerade Anzahl an Einsen vorhanden ist.

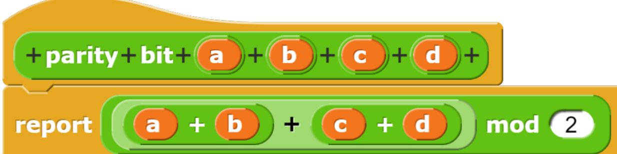



Abbildung 4.9.: Paritätsbit

Beispiele:
1001 enthält zwei Einsen, also ist das Paritätsbit eine Null und wir übertragen 10010. 0111 enthält drei Einsen, also ist das Paritätsbit eine Eins und wir übertragen 01111. Der Empfänger prüft dann, ob in jedem Wort eine gerade Anzahl von Einsen auftritt. Ist die Anzahl ungerade, liegt ein Fehler vor.

### 4.2.4. Größter gemeinsamer Teiler

Für alle positiven ganzen Zahlen a, b, k mit a >b gilt: Wenn k Teiler von a und Teiler von b ist, dann ist k auch Teiler der Differenz a − b. Diese Eigenschaft nutzt der klassische Euklidische Algorithmus aus, um den größten gemeinsamen Teiler (ggT) zweier Zahlen zu berechnen. In moderner Sprache lässt er sich wie folgt formulieren: Gegeben seien zwei natürliche Zahlen a und b (a>b):

**ggT(a, b: Ganzzahl): Ganzzahl**

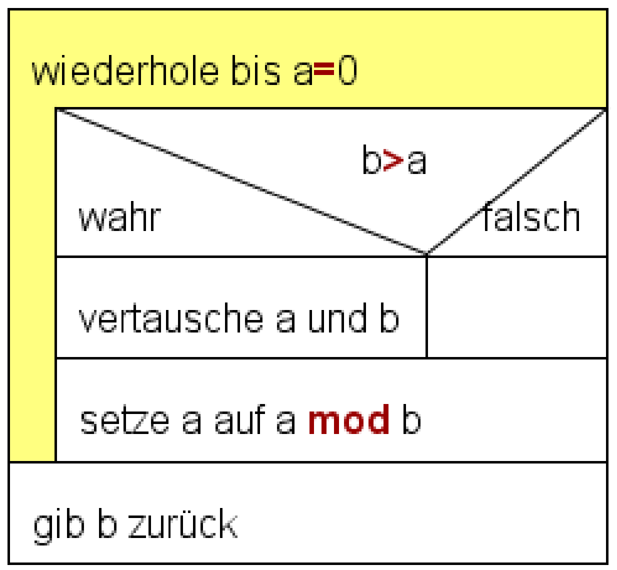



Abbildung 4.10.: moderner Euklidischer Algorithmus

1. Solange a $\neq$ 0, wiederhole die Schritte 2. und 3.
2. Falls b>a, vertausche die Zahlen a und b.
3. Ersetze a durch a - b.
4. (Wenn die Abbruchbedingung a=0 erreicht ist:) Der größte gemeinsame Teiler ist die Zahl b.

Dazu ein Beispiel: Gesucht ist der größte gemeinsame Teiler von a=152 und b=24, $a > b$. $152 - 24 = 128$. Damit wird a=128.

Es wird die nächste Differenz gebildet: $128 - 24 = 104$. Damit wird a=104.

Es wird die dritte Differenz gebildet: $104 - 24 = 80$. Nun ist a=80.

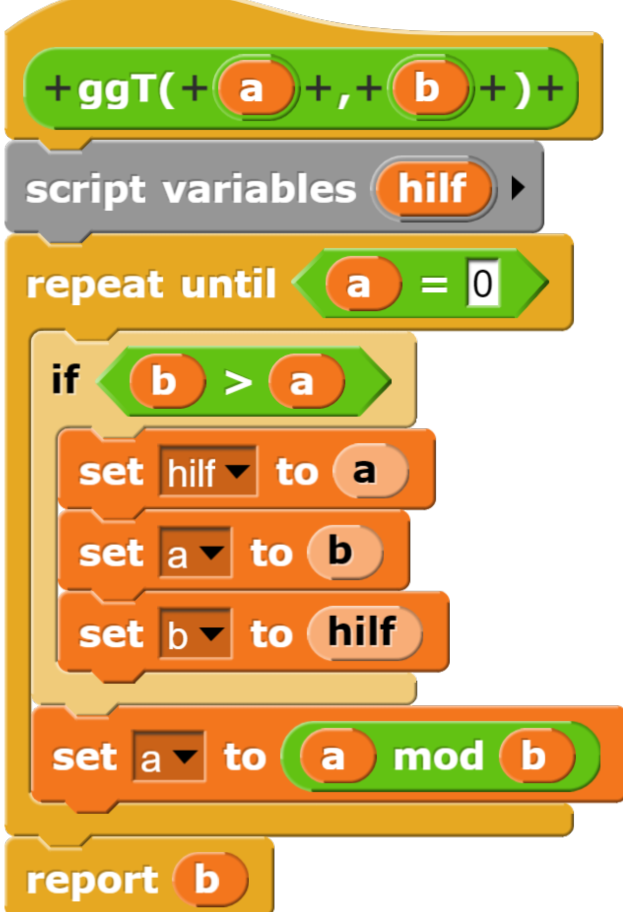



Abbildung 4.11.: Euklidischer Algorithmus

Die vierte Differenz ist $80 - 24 = 56$, a=56.

Die fünfte Differenz ist $56 - 24 = 32$, a=32.

Die sechste Differenz ist $32 - 24 = 8$. Nun ist a=8 und b=24. $b > a$, wir müssen deshalb die Zahlen vertauschen: a=24, b=8.

Die siebte Differenz ist $24 - 8 = 16$. a=16.

Die achte Differenz ist $16 - 8 = 8$. a=8

Die neunte Differenz ist $8 - 8 = 0$. a=0, damit ist die Abbruchbedingung erreicht und

der größe gemeinsame Teiler von 152 und 24 ist 8.

Man erkennt, dass der Algorithmus zum Ziel kommt, aber man sieht auch den größten Nachteil dieses klassischen euklidischen Algorithmus: Man benötigt gegebenenfalls sehr viele Differenzen, um die Abbruchbedingung zu erreichen. Der **moderne Euklidische Algorithmus** nutzt deshalb eine andere Eigenschaft der positiven ganzen Zahlen a, b, k mit $a > b$: Wenn k Teiler von a und Teiler von b ist, dann ist k auch Teiler von `a mod b`. In dem oben notierten Algorithmus wird nur die dritte Zeile durch „Ersetze a durch a mod b" ersetzt.

Auch dazu ein Beispiel: Gesucht ist der größte gemeinsame Teiler von a=221 und b=119, a >b.

221 durch 119 ist 1 Rest 102. Also wird a=102. Nun ist a=102 kleiner als b=119, wir tauschen die beiden Zahlen.

119 durch 102 ist 1 Rest 17. Also wird a=17. Nun ist a=17 kleiner als b=102, wir tauschen die beiden Zahlen.

102 durch 17 ist 6 Rest 0. Wir haben die Abbruchbedingung erreicht. Der größte gemeinsame Teiler ist b=17.

Für einen Tauschvorgang benötigen wir immer einen Zwischenspeicher. Wenn wir die Speicherstelle von b mit dem Inhalt von a überschreiben, geht der alte Inhalt von b verloren. Würden wir jetzt die Speicherstelle von a mit dem Inhalt von b überschreiben, dann hätten wir zwei Mal a.

## 4.3. Wahrheitswert

Wahrheitswerte werden mit dem `SET`-Block eingestellt.

Sie können anschließend überall in alle sechseckigen Lücken eingesetzt werden. Die sechseckigen Lücken stehen für Bedingungen. Dabei erfolgt keine Prüfung durch Snap!. Der Programmierer ist dafür verantwortlich, dass die Variable den Wert true oder false hat.

Ein Wahrheitswert ist bereits eine Bedingung. Sobald z.B. ein `set gefunden to true` erfolgt ist, bedarf ein keiner weiteren Umschreibung. Also heißt es `if gefunden ...` und nicht `if gefunden = true ...` Das letztere

Abbildung 4.12.: Wahrheitswert

ist eine unschöne Formulierung, die zudem den Verdacht weckt, der Programmierer habe die Eigenschaft eines Wahrheitswertes nicht richtig verstanden.

**Achtung:** Der `Change`-Block darf nicht auf Wahrheitswerte angewandt werden! `Change` ist ausschließlich für Zahl-Variable geeignet. Wird in dem oben angegebenen Beispiel `change gefunden by 1` eingegeben, dann setzt Snap! voraus, dass es sich nicht um einen Wahrheitswert, sondern um eine Zahlvariable handelt, setzt den Inhalt auf Null und addiert Eins dazu.

## 4.4. Aufgaben

**Aufgabe 4.1** *Eine Firma gewährt Großhändlern (G) 15%, Einzelhändler (E) 10% und normalen Kunden (K) 5% Rabatt. Gesucht ist ein Block* ***Rechnungsbetrag*** *mit den Parametern Kundentyp (G, E oder K) und Nettobetrag, der den richtigen Rechnungsbetrag liefert. Beachten Sie, dass nach Abzug des Rabatts 19% Mehrwertsteuer hinzu zu addieren sind und dass Geldbeträge mit zwei Nachkommastellen angegeben werden.*

**Aufgabe 4.2** *Gesucht ist ein Block* ***LineareGleichung*** *mit den Parametern a und b, der die Lösung der linearen Gleichung $a \cdot x + b = 0$ ausgibt. Beachten Sie, dass die Gleichung auch keine reelle Lösung haben kann. In diesem Fall soll der Block false ausgeben.*

**Aufgabe 4.3** *Gesucht ist ein Block* ***Summe****, der die Summe aller natürlichen Zahlen zwischen A und B berechnet, wobei A und B Parameter des Blocks sind. Beispiel A=5, B=10, 5+6+7+8+9+10=45. Sie können dabei voraussetzen, dass $A < B$.*

**Aufgabe 4.4** *In jedem Dreieck ist die Summe der Längen zweier Seiten größer als die der dritten Seite. Gesucht ist ein Predicate-Block **isTriangle** mit den Parametern a, b und c, der prüft, ob drei Zahlen diese Bedingungen erfüllen, also ob sie die Seiten eines Dreiecks bilden können.*

**Aufgabe 4.5** *Zu einer gegebenen natürlichen Zahl soll die Zerlegung in Primfaktoren bestimmt werden. Beispiel:* $11725 = 5^2 \cdot 7 \cdot 67$ *Dazu findet sich im Internet der nebenstehende Algorithmus:*

**primfaktoren (n: Ganzzahl): Liste**

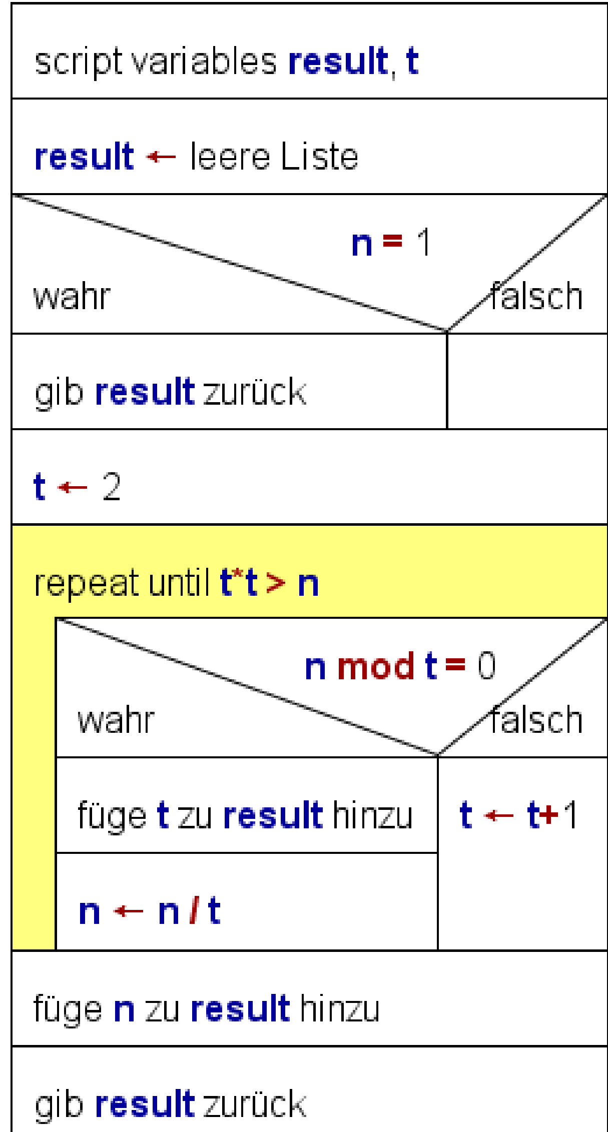



Abbildung 4.13.: zu Aufgabe 4.5 und 4.6 Primfaktorzerlegung

*Implementieren Sie den Algorithmus in Snap!.*

**Aufgabe 4.6** *Erstelle eine Tracetabelle für die Berechnung der Primfaktorzerlegung von 84 nach dem Algorithmus aus 4.5*

**Aufgabe 4.7** *Zu einer gegebenen natürlichen Zahl sollen alle Teiler ermittelt werden. Implementiere eine Operation, die eine natürliche Zahl als Parameter erhält und die zugehörigen Teiler als Liste ausgibt.*

*Beispiel: Teiler von 6 sind 1, 2, 3, 6.*

**Aufgabe 4.8** *Erstelle eine Tracetabelle zum ggT(17262, 8580). Der dazugehörige Algorithmus findet sich in Kapitel 2.3.*

**Aufgabe 4.9** *Beschreiben Sie einen Algorithmus zur Berechnung des Wertes einer Zahl, deren Dezimalziffern im ASCII-Code angegeben sind, mit Hilfe eines Struktogramms. Hinweis: Die ASCII-Codes der Ziffern liegen zwischen 0 (48) bis 9 (57). Beispiel: 49 57 56 52 ergibt 1984, wobei die Leerstellen nur zur besseren Lesbarkeit eingesetzt wurden. Die ursprüngliche Eingabe war 49575652.*

*Erstellen Sie eine Tracetabelle zu einer beliebigen vierstelligen Dezimalzahl, deren Ziffern im ASCII-Code angegeben sind, zur Überprüfung Ihrer Lösung.*

**Aufgabe 4.10** *Beschreiben Sie einen Algorithmus zur Berechnung des Wertes einer Zahl, deren Hexadezimalziffern im ASCII-Code angegeben sind, mit Hilfe eines Struktogramms. Der Wert soll als Dezimalzahl ausgegeben werden.*

*Erstellen Sie zur Kontrolle Ihrer Operation eine Tracetabelle zu einer vierstelligen Zahl mit vier verschiedenen Ziffer, wobei zwei Ziffern aus dem Bereich von 0 bis 9 und zwei Ziffern aus dem Bereich von A bis F gewählt werden sollen.*

**Aufgabe 4.11** *Die EAN (European Article Number) taucht auf vielen Produkten auf. Sie besteht aus einer Zahl mit zwölf Stellen und einer Prüfziffer, die hinten angehängt wird. Die Prüfziffer wird nach folgenden Regeln berechnet:*

1. *Von links nach rechts werden die Stellen abwechselnd mit 1 und 3 gewichtet.*

2. *Die jeweiligen Produkte aus der Stelle und der Zahl 1 bzw. 3 werden errechnet und summiert.*

3. *Die Prüfziffer ist der volle Rest zur nächsthöheren durch 10 teilbaren Zahl (Modulo 10). Falls die Summe durch 10 teilbar ist, ist die Prüfziffer 0.*

*Beispiel: Die Zahl 123456789012 liefert im Schritt 2*

$1 \cdot 1 + 2 \cdot 3 + 3 \cdot 1 + 4 \cdot 3 + 5 \cdot 1 + 6 \cdot 3 + 7 \cdot 1 + 8 \cdot 3 + 9 \cdot 1 + 0 \cdot 3 + 1 \cdot 1 + 2 \cdot 3 = 1+6+3+12+5+18+7+24+9+0+1+6 = 92$
*Da* $92 + 8 = 100$ *und* $mod(100, 10) = 0$*, ist die Prüfziffer 8.*

*Gesucht ist ein Block* ***EAN****, der zu einer Zahl die Zahl mit EAN-Prüfziffer ausgibt.*

# 5. Dualzahlen

In diesem Kapitel lernen Sie

- wie Dualzahlen aufgebaut sind
- wie man Dezimalzahlen in Dualzahlen umwandelt und umgekehrt,
- wie man mit Dualzahlen rechnet,
- warum der Computer Probleme mit Fünfteln hat und mit wie viel Stellen er rechnet,
- wie Hexadezimalzahlen aufgebaut sind,
- wie man Dezimalzahlen in Hexadezimalzahlen umwandelt und
- wie man zwischen der Darstellung einer Zahl in dualer bzw, hexadezimalzahler Schreibweise wechselt.

## 5.1. Grundlagen

Welche Sprache sprechen eigentlich Computer? Viele Worte aus ihrem Umfeld stammen offenbar aus dem Englischen, aber bei der Installation eines neuen Betriebssystems lassen sich auch andere Sprachen einstellen, z.B. Deutsch. Aber damit ein Computer einen Befehl wie „speichere“ oder „load“ versteht, muss dieser Befehl übersetzt werden. Die eigentliche Sprache von Computern nennt man **Maschinensprache**. Sie besteht aus einer Folge von Nullen und Einsen. Diese Befehle sind für Menschen nur schwer lesbar. Wir verwenden deshalb menschenlesbare Programmiersprachen (man nennt sie auch **höhere Programmiersprachen**), die durch spezielle Programme dann in Maschinensprache übersetzt werden.

Computer rechnen auch mit solchen Folgen aus Nullen und Einsen, die wir **Dualzahlen** nennen. Der Begriff „Dual“ deutet dabei an, dass es in diesem Zahlensystem nur zwei verschiedene Ziffern gibt, nämlich 0 und 1.

Um zu verstehen, wie Dualzahlen aufgebaut sind, muss man sich zunächst die Struktur unserer bekannten Dezimalzahlen klar machen. Wir verwenden in der Mathematik ein stellenbasiertes Zahlensystem. Die Zahl 137 setzt sich zusammen aus einem Hunderter, drei Zehnern und sieben Einern. Generell gilt in einem stellenbasierten Zahlensystem: Der Wert einer Ziffer hängt von deren Platz ab. So stehen an der letzten Stelle (bzw. an der Stelle vor dem Komma) immer die Einer. Das Dezimalsystem verfügt über die zehn Ziffern von 0 bis 9. Die kleinste natürliche Zahl ist 0. Die darauf folgenden Zahlen werden mit den übrigen Ziffern (1 bis 9) dargestellt. Wenn der Vorrat an Ziffern erschöpft ist (bei der 9), dann wird eine neue Stelle hinzugefügt, die Zehnerstelle. Mit Hilfe der Einer- und der Zehnerstelle können 100 verschiedene Zahlen (von 0 bis 99) dargestellt werden. Dann sind die Kombinationsmöglichkeiten erschöpft und wir nehmen eine dritte Stelle hinzu, die Hunderterstelle. Das Verfahren wird fortgesetzt, wobei wir jeweils die nächste Zehnerpotenz als Basis verwenden, also 1, 10, 100, 1000 usw. Auf diese Weise lassen sich mit nur 10 Ziffern beliebig große Zahlen darstellen. Das heute verwendete Dezimalsystem hat viele Vorteile, die zu seiner großen Verbreitung beigetragen haben. Es gibt in bestimmten Nischen noch Überreste anderer Zahlensysteme, z.B. die römischen Zahlen, bei denen die einzelnen Buchstaben unterschiedliche Werte haben. So bedeutet **MMXXII** 2022, weil M für Tausend, X für Zehn und I für Eins steht. Bei römischen Ziffern entscheidet der Buchstabe und nicht die Stelle über den Wert. Bei Zeit- und Winkelangaben finden wir noch Reste des babylonischen Zahlensystems , etwa in der Unterteilung des Kreises in 360 Grad oder der Stunde in 60 Minuten. Das babylonische Zahlensystem beruhte auf der 60.

Die Dualzahlen und damit die Computerzahlen sind ebenfalls ein stellenbasiertes Zahlen-

system. Wir haben aber nur zwei Ziffern zur Verfügung, 0 und 1. Das liegt daran, dass der Computer nur unterscheiden kann, ob auf einer Leitung Strom fließt oder kein Strom fließt. Um Dezimal- und Dualzahlen in einem gemischten Kontext unterscheiden zu können, markieren wir die Dualzahlen mit einer tiefgestellten Zwei (für die Basis 2). Die letzte Stelle ist wiederum die Einerstelle. Die kleinste Dualzahl ist $0_2 = 0$. Dann folgt die $1_2 = 1$. Für 0 und 1 sind also duale und dezimale Schreibweise gleich. Mit der 1 sind im Dualsystem alle Möglichkeiten erschöpft. Als Basis können wir natürlich nicht 10 verwenden (dann bräuchten wir ja 10 Ziffern, wir haben aber nur 2), sondern 2. Die Dualzahlen bauen also auf den **Zweierpotenzen** auf. Wir haben Einer, Zweier, Vierer, Achter, usw. Nach der $1_2$ brauchen wir eine neue Stelle für die Zweier. Es folgen $10_2 = 2$ und $11_2 = 3$. Wieder sind alle Möglichkeiten erschöpft und wir führen eine dritte Stelle ein für die Vierer. Es folgen die Zahlen $100_2 = 4$, $101_2 = 5$, $110_2 = 6$ und $111_2 = 7$. Die vierte Stelle steht für die Achter. Damit lassen sich die Zahlen von $1000_2 = 8$ bis $1111_2 = 15$ darstellen. Die folgenden Stellen stehen für die Sechszehner, Zweiunddreißiger, Vierundsechziger usw.

Wir notieren die Dualzahlen von 0 bis 15 in einer Tabelle, weil diese häufig vorkommen:

| | | |
|---|---|---|
| $0000_2 = 0$ | $0001_2 = 1$ | $0010_2 = 2$ |
| $0011_2 = 3$ | $0100_2 = 4$ | $0101_2 = 5$ |
| $0110_2 = 6$ | $0111_2 = 7$ | $1000_2 = 8$ |
| $1001_2 = 9$ | $1010_2 = 10$ | $1011_2 = 11$ |
| $1100_2 = 12$ | $1101_2 = 13$ | $1110_2 = 14$ |
| $1111_2 = 15$ | | |

Dualzahlen notieren wir häufig mit führenden Nullen, weil Computer eine feste Anzahl von parallelen Leitungen verwenden, um eine Dualzahl zu übertragen oder abzuspeichern. Wenn ich auf acht Leitungen die Dualzahl $11_2 = 3$ übertragen möchte, müssen die beiden niederwertigsten Leitungen Eins sein, aber alle anderen 6 Leitungen müssen den Wert Null haben, sonst könnte ich die Zahl z.B. mit $00001011_2 = 11$ verwechseln. Ich muss also $00000011_2$ übertragen.

Die einzelnen Stellen einer Dualzahl heißen **Bits**. Das ist eine Abkürzung des englischen „binary digit“ und bedeutet soviel wie binäre Ziffer. Binär bezeichnet etwas, das zwei Zeichen annehmen kann, also nicht nur Zahlen. Unter Dual versteht man ein Zahlensystem, das nur aus zwei Zeichen besteht.

Acht Bits zusammengefasst bilden ein **Byte**, also eine achtstellige Dualzahl. Speicherangaben erfolgen als Vielfache von Byte, z.B. KB (Kilobyte = 1000 Byte), MB (Megabyte = 1 Million Byte) und GB (Gigabyte = 1 Milliarde Byte)

Es gibt ein einfaches Hilfsmittel, um mit den Dualzahlen besser vertraut zu werden. Man nimmt einen dünnen Streifen Papier oder Karton und schneidet davon Stücke unterschiedlicher Länge ab, nämlich 1 cm, 2 cm, 4 cm, 8 cm, 16 cm. Jede Dualzahl zwischen 0 und 31 lässt sich mit diesen fünf Stücken darstellen. Mit Hilfe eines Lineals kann man dann ein einfaches Spiel spielen: Einer nennt eine Zahl zwischen 0 und 31 (wenn man keinen Mitspieler hat, kann der Taschenrechner oder der Computer eine solche Zahl liefern, z.B. mit `pick random 0 to 31`). Der andere versucht dann, mit den fünf Stücken diese Zahl zu legen.

## 5.2. Dezimal nach Dual

Wenn wir mit Dualzahlen rechnen wollen, müssen wir die Rechenergebnisse auch überprüfen. Dazu müssen wir Dezimalzahlen in Dualzahlen umwandeln und umgekehrt. Wir beginnen mit der Umwandlung von Dezimalzahlen in Dualzahlen.

Dafür gibt es zwei mögliche Vorgehensweisen: Wir fangen bei der größten (von Null verschiedenen) Ziffer an oder bei der kleinsten. Für kleine Dualzahlen spielt die Richtung keine Rolle, weil wir uns schnell an Achter, Vierer, Zweier und Einer gewöhnen. Aber bei größeren Zahlen hat man die Zweierpotenzen nicht aus dem Gedächtnis parat, z.B. $2^{15} = 32768$ und auch die Differenz $n - 32768$ lässt sich nicht einfach im Kopf berechnen. Deshalb beginnen wir von hinten, also mit der Einerziffer.

Die Einerziffer ist Null, wenn die Dezimalzahl gerade ist, und Eins, wenn die Dezimalzahl ungerade ist. Woran liegt das? Denken wir an das Spiel zurück: Bei unseren fünf Stücken gibt

es nur ein Stück mit einer ungeraden Länge, nämlich die Eins. Alle anderen Stücke haben eine gerade Länge. Die Summe zweier geraden Zahlen ist immer gerade. Das gilt auch für alle höheren Zweierpotenzen, die ja aus der Multiplikation einer bestimmten Anzahl von Zweien entstehen. Alle ungeraden Zahlen enthalten also genau ein Mal die Eins und alle Zahlen, die eine Eins enthalten, sind ungerade. Jedes Stück im Spiel darf ja nur ein Mal in einer Zahl auftauchen.

Die Einerziffer der Dualzahl könnte man also einfach an der letzten Ziffer der Dualzahl ablesen: Bei Endziffer $0, 2, 4, 6, 8$ ist die Einerziffer 0, bei Endziffer $1, 3, 5, 7, 9$ ist die Einerziffer 1.

Um die Umwandlung zu automatisieren und damit für Computer handhabbar zu machen, greifen wir auf ein Verfahren zurück, dass wir in der Grundschule kennen gelernt haben, die Division mit Rest. Im Unterschied zur Addition und Multiplikation ergibt die Division zweier natürlicher Zahlen ja nur in Ausnahmefällen wieder eine natürliche Zahl, z.B. $18 : 6 = 3$. Bei der Division mit Rest gilt z.B. $20 : 6 = 3$ Rest 2.

Die Division mit Rest ist in der Informatik viel wichtiger als in der Mathematik, wo sie in Klasse 3 eingeführt und dann schnell wieder beiseite gelegt wird. Es gibt sogar einen eigenen Befehl für den Rest, nämlich `MOD`. Die letzte Dualziffer von `zahl` erhalten wir also mit `zahl mod 2`. Der Befehl für die ganzzahlige Division `DIV` ist in Snap! leider nicht enthalten. Wir können ihn aber einfach ersetzen, indem wir `zahl` durch Zwei dividieren und dann mit `floor` abrunden: `floor of (zahl/2)`.[1]

Damit kommen wir an die nächste Ziffer der Dualzahl heran. Damit ergibt sich das folgende schriftliche Verfahren für die Umwandlung einer Dezimalzahl in eine Dualzahl.

`Wiederhole den folgenden Schritt, bis die Dezimalzahl Null ist:`

`Notiere den Rest bei der ganzzahligen Division durch 2 und teile die Dezimalzahl ganzzahlig durch 2.`

`Dann ergeben die Reste von unten nach oben geschrieben die Dualzahl.`

Beispiel (mit 22):
22 durch 2 ist 11 Rest 0.
11 durch 2 ist 5 Rest 1.
5 durch 2 ist 2 Rest 1.
2 durch 2 ist 1 Rest 0.
1 durch 2 ist 0 Rest 1.
Damit ist die Abbruchbedingung erfüllt. Die Reste von unten nach oben gelesen ergeben die Dualzahl: $22 = 10110_2$.

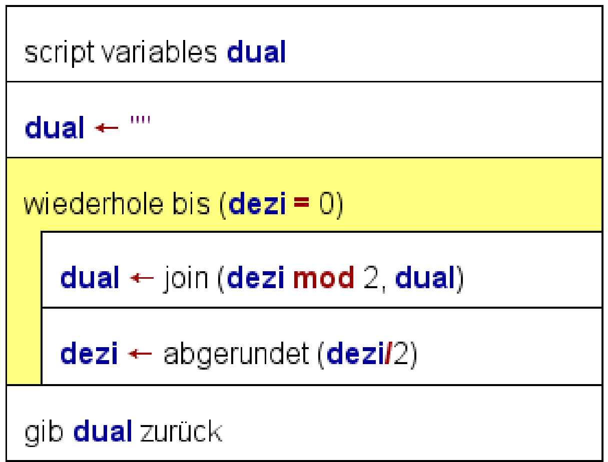



Abbildung 5.1.: Dezimalzahlen in Dualzahlen

Wenn wir dieses Verfahren in Snap! übersetzen, müssen wir drei Dinge beachten:

Bevor wir die Skriptvariable `dual` verwenden können, müssen wir sie zunächst mit `set dual to '' ''` löschen. Das liegt daran, dass alle Variablen bei Einrichtung den Wert Null erhalten. Wird das Löschen von `dual` vergessen, so steht vor unserer Dualzahl eine zusätzliche Null. 22 würde dann zu $010110_2$.

Die Restzahlen erhalten wir jeweils als Rest bei der Division durch 2, also mit `dezi mod 2`. Damit wir sie am Ende nicht umdrehen müssen, hängen wir sie schrittweise jeweils **vorn** am Ergebnis an. Auf diese Weise steht der erste Rest ganz hinten und der letzte Rest ganz vorn.

Den Wert von `dezi` müssen wir in jedem Schritt halbieren. Bei ungeraden Zahlen tritt ein Nachkommawert von 0,5 auftauchen, deshalb wird jeweils mit `floor of ZAHL` abgerundet.

[1] Alternativ können wir für den DIV-Befehl auch den Rest bei Division durch b abziehen und dann teilen: `a div b = (a - (a mod b)) / b`, z.B. 17 mod 3 = 2; (17 - 2)/3 = 5

```
+deziToDual(+ dezi # +)+
script variables dual
set dual to
repeat until dezi = 0
    set dual to join (dezi mod 2) dual
    set dezi to floor of (dezi / 2)
report dual
```

Abbildung 5.2.: Dezimal nach Dual

## 5.3. Dual nach Dezimal

Um eine als Zeichenkette gegebene Dualzahl in die zugehörige Dezimalzahl umzuwandeln, muss jede Ziffer mit der zugehörigen Zweierpotenz multipliziert werden. Die Summe der Produkte ergibt dann die Dezimalzahl. Das Verfahren kann man am besten durch eine Tabelle veranschaulichen.

Wir wählen wieder unser Beispiel $10110_2$ und tragen die Dualziffern in die mittlere Zeile einer Tabelle ein:

| 1 | 0 | 1 | 1 | 0 | |
|---|---|---|---|---|---|

In die obere Zeile tragen wir die Zweierpotenzen ein, rechts beginnend mit $2^0 = 1$:

| 16 | 8 | 4 | 2 | 1 | |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | |

In die untere Zeile tragen wir das Produkt der beiden oberen Zellen ein und addieren diese Produkte dann:

| 16 | 8 | 4 | 2 | 1 | |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | |
| 16 | 0 | 4 | 2 | 0 | $16 + 4 + 2 = 22$ |

Soweit die händische Form.

Für das Programm benötigen wir eine Skriptvariable für den Wert der Potenz. Bei einer einstelligen Dualzahl stehen vorn die Einer, also $2^0 = 1$. Bei einer zweistelligen Dualzahl stehen vorn die Zweier, also $2^1 = 2$. Bei einer dreistelligen Dualzahl stehen vorn die Vierer, also $2^2 = 4$. Allgemein gilt: Bei einer n-stelligen Dualzahl steht vorn die Potenz $2^{n-1}$.

**dualToDezi (dual: Zeichenkette): Ganzzahl**

```
script variables potenz, dezi, i
dezi ← 0
potenz ← 2 ^ (Länge dual - 1)
für i von 1 bis Länge(dual)
    dezi ← dezi + dual[i] * potenz
    potenz ← potenz / 2
gib dezi zurück
```

Abbildung 5.3.: Dual nach Dezimal

Die eigentliche Umrechnung erfolgt dann mit einer Zählschleife. Wir gehen die Dualzahl ziffernweise durch. Die aktuelle Ziffer erhalten wir mit `letter i of dual`. In jedem Schritt wird der aktuelle Wert der Potenz mit der aktuellen Dualziffer multipliziert und das Ergebnis zu **dezi** hinzu addiert. Anschließend wird die Potenz durch 2 dividiert, um den nächsten Wert der Potenz zu erhalten.

```
+dualToDezi(+ dual +)+
script variables potenz dezi i
set dezi to 0
set potenz to (2 ^ ((length of text dual) − 1))
for i = 1 to length of text dual
    change dezi by ((letter i of dual) × potenz)
    set potenz to (potenz / 2)
report dezi
```

Abbildung 5.4.: Dual nach Dezimal in Snap!

## 5.4. Rechnen mit Dualzahlen

Die Addition von zwei Dualzahlen können wir auf die bekannte schriftliche Addition von natürlichen Zahlen zurückführen. Allerdings ist die Zahl der Rechenregeln viel kleiner, weil wir ja nur zwei Ziffern haben. Zunächst gibt es nur vier Regeln: **0+0=0**, **0+1=1**, **1+0=1** und **1+1=0 Übertrag 1**, Bei der Addition von 1 und 1 entsteht ein Übertrag, weil wir die dezimale Zwei dual als $10_2$ darstellen. Wir benötigen also auch Regeln für das Rechnen mit einem Übertrag:
**0+0+1=1**,
**0+1+1=0 Übertrag 1**,
**1+0+1=0 Übertrag 1** und
**1+1+1=1 Übertrag 1**.

In der folgenden Beispielrechnung werden alle acht Regeln benötigt:

| | | |
|---|---|---|
| 1. Summand | | 01101100 |
| 2. Summand | + | 01011010 |
| Übertrag | | 11110000 |
| | = | 11000110 |

Wir könnten jetzt entsprechende Regeln für die schriftliche Subtraktion formulieren, aber im Hinblick auf die Durchführung von Rechnungen in einem Computer gehen wir einen anderen Weg. Statt einen Computer mit einem Addier- und einem Subtrahierwerk auszurüsten, wäre es doch einfacher, eine einzige, umschaltbare Addier-/Subtrahierschaltung zu bauen. Um zu zeigen, wie solch eine Schaltung funktionieren kann, gehen wir wieder den Umweg über das Dezimalsystem.

Wir nehmen eine beliebige natürliche Zahl, z.B. 987.654.321, und bilden das Neunerkomplement. Das Neunerkomplement hat genau so viele Stellen wie unsere ursprüngliche Zahl (9) und jede einzelne Ziffer des Neunerkomplement ergibt mit der gleichen Ziffer der urspünglichen Zahl addiert genau Neun. Das Neunerkomplement von 987.654.321 ist also 012.345.678. Im zweiten Schritt bilden wir das Zehnerkomplement unserer ursprünglichen Zahl, indem wir zum Neunerkomplement Eins hinzu addieren. Das Zehnerkomplement wäre also 012.345.679.

Nun kommt der Clou: Wir können eine Dezimalzahl subtrahieren, indem wir ihr Zehnerkomplement addieren und im Ergebnis die erste Stelle streichen. Dazu ein Beispiel:

| | |
|---|---|
| | 990.000.000 |
| - | 987.654.321 |
| = | 2.345.679 |

| | |
|---|---|
| | 990.000.000 |
| + | 012.345.679 |
| = | 1.002.345.679 |

Streichen wir die erste Stelle der Addition des Zehnerkomplements, dann erhalten wir das Ergebnis der gesuchten Subtraktion. Warum funktioniert die Rechnung? Die Summe aus einer Zahl und ihrem Neunerkomplement ergibt eine Zahl aus lauter Neunen. Addieren man 1 dazu, dann erhält man eine Zahl mit einer Eins am Anfang und lauter Nullen. Genau diese Eins streichen wir im Ergebnis weg.

Um eine Dualzahl zu subtrahieren, bilden wir zunächst das Einerkomplement. Die Ziffern des Einerkomplements werden so gewählt, dass die Summe der beiden Ziffern Eins ergibt. Die Dualzahl 1100 hat das Einerkomplement 0011. Das Zweierkomplement erhalten wir, indem wir Eins hinzuaddieren. Das Zweierkomplement ist 0100.

| | | |
|---|---|---|
| | 1111 | 15 |
| - | 1100 | 12 |
| = | 11 | 3 |

| | | |
|---|---|---|
| | 1111 | Minuend |
| + | 0100 | Zweierkomplement des Subtrahenden |
| = | 10011 | |

Auch hier erhalten wir das Ergebnis der Subtraktion, wenn wir bei der Addition des Zweierkomplements die erste Stelle streichen.

Leider können wir mit den Dualzahlen nicht direkt rechnen, wenn wir sie als Zeichenketten speichern. Zum Rechnen müssen wir also

entweder die Dualzahlen in Dezimalzahlen umwandeln, dann die Rechenoperation ausführen und das Ergebnis wieder in eine Dualzahl umwandeln. Oder wir schreiben einen eigenen Block für die Addition zweier Dualzahlen und einen zweiten für die Bildung der Zweierkomplements.

## 5.5. Wechselgeldautomat

Kern eines Wechselgeldautomaten ist ein Block namens `wechselgeld` mit den Parametern `preis` und `gegeben`. Die Differenz zwischen `gegeben` und `preis` soll kleiner sein als 5€. Der Block soll das Wechselgeld mit möglichst wenig Münzen zu der Differenz berechnen und zunächst in Form eines Liste ausgeben (wir werden abschließend die Ausgabe etwas benutzerfreundlicher machen).

Wir machen uns zunächst an einem Beispiel klar, wie der Automat funktioniert: Wir kaufen einen Fahrkarte zum Preis von 3,20 € und bezahlen mit einem 5-€-Schein. Die Differenz zwischen 5 € und 3,20 € beträgt 1,80 €. Der Automat gibt zunächst ein 1-€-Stück zurück (2 € wären zu viel). Dann fehlen noch 80 ct. Also gibt er ein 50-ct-Stück zurück. Es bleibt eine Restdifferenz von 30 ct, die durch ein 20-ct-Stück und ein 10-ct-Stück ausgeglichen werden.

Wie oft können die einzelnen Geldstücke in einer Menge mit einer minimalen Anzahl von Münzen auftauchen? Wir beginnen mit dem kleinsten Geldstück, der 1-ct-Münze. Wenn wir die 1-ct-Münze verdopppeln, erhalten wir 2 ct. Dafür gibt es eine eigene Münze. In einer minimalen Menge kann also höchstens ein 1-ct-Stück enthalten sein. Gäbe es zwei solcher Münzen, dann könnte man sie durch ein 2-ct-Stück ersetzen und hätte eine Münze weniger. Wenn wir die 2-ct-Münze verdoppeln, gibt das 4 ct. Dafür gibt es aber keine eigene Münzen. Also können in einer minimalen Menge bis zu 2 2-ct-Münzen erhalten sein, weil das nächste Geldstück, die 5-ct-Münze, kleiner ist als drei mal 2 ct. Verdoppeln wir die 5-ct-Münze, ergeben sich 10 ct und damit die nächste Münze. Eine 5-ct-Münze kann in unserer minimalen Menge also nur ein Mal enthalten sein. Das gleiche gilt für 10 ct, 50 ct und 1 €. 20 ct und 2 € können aber maximal zwei Mal in der Menge enthalten sein, weil sie weniger als halb so groß sind wie die nächstgrößere Münze.

Damit können wir den Algorithmus für unseren Wechselgeldautomaten formulieren: Wir erzeugen eine Liste mit den möglichen Geldstücken unter Beachtung der Häufigkeit, also `set wgeld to list{2; 2; 1; 0.5; 0.2, 0.2, 0.1; 0.05; 0.02; 0.02; 0.01` } Dann berechnen wir die Differenz zwischen Gegeben und Preis und prüfen - vorn beginnend - für jedes Geldstück, ob die Differenz noch größer oder gleich dem Geldstück ist. Falls ja, fügen wir das Geldstück zum Ergebis hinzu und ziehen seinen Betrag von der Differenz ab.

```
+wechselgeld(+gegeben #+,+preis #+)+
script variables result i wgeld differenz
set result to list
set wgeld to list 2 2 1 0.5 0.2 0.2 0.1 0.05 0.02 0.02 0.01
set differenz to gegeben − preis
for i = 1 to length of wgeld
    if differenz > item i of wgeld or differenz = item i of wgeld
        add item i of wgeld to result
        set differenz to differenz − item i of wgeld
report result
```

Abbildung 5.5.: Wechselgeldautomat

Wenn wir den Block systematisch testen, also mit einer Reihe von Preisen, die in Cent-Schritten aufeinander folgend (z.B. 3,80; 3,81; 3,82; 3,84), stellen wir fest, dass er nicht immer richtig funktioniert. Bei jeder zweiten Rechnung fehlt ein Cent beim Wechselgeld. Woran kann das liegen?

Die einzige Größe außer der Zählvariable, die sich in der `FOR`-Schleife verändert, ist die Differenz. Um besser zu sehen, was dort passiert, speichern wir in jedem Schritt statt des Wechselgelds die Differenz in Result und am Schluss noch den letzten Wert der Differenz. Für die Eingabe `wechselgeld(5; 3.85)` erhalten wir die Differenzen { 1,15; 0,14999999999; 0,0499999999906; 0,0299999999905; 0,0099999999905 }. Man

sieht sehr schnell, warum 1 ct in der Abrechnung fehlt: 0,009 ist kleiner als 0,01. Auch das 5-ct-Stück, das neben dem 1 € und den 10 Cent in der richtigen Lösung enthalten wäre, wird knapp verfehlt, weil der Computer statt 0,15-0,10=0,05 rechnet: 0,14999999999-0,1=0,04999999999096, was kleiner ist als 0,05. Warum kann der Computer eine einfache Rechnung wie 0,15-0,1=0,05 nicht ausführen?

Wir kennen das Problem von den Dezimalzahlen. Alle Brüche, bei denen in der Primfaktorzerlegung des Nenner andere Primzahlen als 2 und 5 auftauchen, also z.B. ein Drittel, führen auf periodische Dezimalzahlen, während die Brüche, deren Nenner nur 2 und/oder 5 sowie deren Vielfache umfasst, auf endliche Dezimalzahlen führen. Deshalb lassen sich Zwanzigstel mit drei Nachkommastellen darstellen, während Fünfzehntel auf eine periodische Dezimalzahl führen.

Die Sprache des Computers sind die Dualzahlen. Alle Brüche, deren Nenner Zweierpotenzen sind, lassen sich als endliche Dualzahlen darstellen. Alle Brüche, in deren Nenner andere Primfaktoren als 2 vorkommen, führen auf periodische Dualzahlen, wie z.B. zwei Fünftel. Periodische Dualzahlen kann der Computer aber nicht speichern. Er berechnet eine Näherungszahl, die so lang ist wie Nachkommastellen vorgesehen sind, und speichert diese ab.

Es gibt zwei verschiedene Möglichkeiten, das Problem zu beheben: Entweder runden wir die Ergebnisse nach jedem Rechenschritt in Zahlen mit zwei Nachkommastellen um (vgl. dazu Kapitel 5.1) oder wir rechnen intern mit Cent und geben die Ergebnisse in Euro und Cent aus.

Wir haben folgende Änderungen an dem Wechselgeldautomaten vorgenommen: Gegeben und Preis rechnen wir durch Multiplikation mit 100 in Cent um. Unsere Wechselgeldliste enthält nicht mehr den Nennwert der Münzen, sondern ihren Centbetrag, also 200 statt 2. Zusätzlich haben wir eine neue Liste `wgeldiB` (Wechselgeld in Buchstaben) für die Ausgabe mit den Währungseinheiten angelegt. Zu 200 (bisher: 2) passt das Element **2€,**, zu 10 das Element **10ct,** Beim Verringern der Differenz rechnen wir mit den Zahlen der `wgeld`-Liste, aber in das Ergebnis packen wir die Elemente

Abbildung 5.6.: Verbesserter Wechselgeldautomat

der `wgeldiB`-Liste. Schließlich wandeln wir Result mit einem `join` in eine Textzeile um, so dass wir ein schöneres Ergebnis erhalten:

Abbildung 5.7.: Ausgabe Wechselgeld

## 5.6. Hexadezimalzahlen

Dualzahlen sind zwar für Computer einfach zu lesen, aber längere Dualzahlen sind für Menschen schlecht lesbar. Die 99 benötigt im Dezimalsystem nur zwei Ziffern, aber im Dualsystem sind es sieben Ziffern: $1100011_2$. Man könnte nun größere Zahlen im Dezimalsystem darstellen, aber dabei verschenkt man Speicherplatz. Um die 10 Ziffern von Null bis Neun darzustellen, benötigt man im Dualsystem vier Stellen. Aber mit vier Stellen könnte man sogar 16 verschiedene Ziffern darstellen. Dann ließen sich nicht nur die Zahlen bis 99, sondern bis 255 mit zwei Ziffern schreiben. Diese Überlegung führ zum Zahlensystem mit der Basis 16, zum Hexadezimalsystem. Da wir für ein Hexadezimalsystem 16 verschiedene Ziffern benötigen, verwenden wir zusätzlich zu den Dezimalziffern die Buchstaben A (=10) bis F (=15) als Hexadezimalziffern.

Im Hexadezimalsystem stehen an der letzten Stelle die Einer, an der vorletzten Stelle die

| hexadezimal | dual | dezimal |
|---|---|---|
| 0 | 0000 | 0 |
| 1 | 0001 | 1 |
| 2 | 0010 | 2 |
| 3 | 0011 | 3 |
| 4 | 0100 | 4 |
| 5 | 0101 | 5 |
| 6 | 0110 | 5 |
| 7 | 0111 | 7 |
| 8 | 1000 | 8 |
| 9 | 1001 | 9 |
| A | 1010 | 10 |
| B | 1011 | 11 |
| C | 1100 | 12 |
| D | 1101 | 13 |
| E | 1110 | 14 |
| F | 1111 | 15 |

```
deziToHex( dec # )
script variables result ziffer
set result to
repeat until dec = 0
    set ziffer to dec mod 16
    if ziffer > 9
        set result to join unicode letter 2 of ziffer + 65 as letter result
    else
        set result to join ziffer result
    set dec to (dec − ziffer) / 16
report result
```

Abbildung 5.8.: Dezimal nach Hexadezimal

Sechzehner, an der drittletzten Stelle die ($16^2 = 256$) Zweihundertsechsundfünfziger usw.

Zur Umrechnung einer Dezimalzahl in eine Hexadezimalzahl verwenden wir eine leicht abgeänderte Version des Algorithmus, den wir zur Umrechnung von Dezimalzahlen in Dualzahlen benutzt haben (siehe Kapitel 6.2):

Wiederhole, bis die Dezimalzahl gleich Null ist:

Notiere den Rest als Ziffer bei der ganzzahligen Division durch 16 und teile die Dezimalzahl ganzzahlig durch 16. Dann ergeben die Ziffern von unten nach oben geschrieben die Hexadezimalzahl.

Beispiel:
317 durch 16 ist 19 Rest 13 = $D_{16}$.
19 durch 16 ist 1 Rest $3_{16}$.
1 durch 16 ist 0 Rest $1_{16}$.
Damit ist die Abbruchbedingung erfüllt. Die Dezimalzahl 317 ist im Hexadezimalsystem $13D_{16}$.

Weil die Reste hier auch zweistellig sein können, benötigen wir eine Fallunterscheidung. Ist die Ziffer größer als 9, dann müssen wir den entsprechenden Buchstaben anfügen, d.h. für Ziffer 10 das A (Unicode 65), für Ziffer 11 das B (Unicode 66) usw. bis Ziffer 15 das F (Unicode 70). Wir erhalten also den Unicode des Buchstabens im Hexadezimalsystem, wenn wir 65 zu der zweiten Zahl der Ziffer addieren, also bei Ziffer 10 unicode(0+65) as Letter gibt A, bei Ziffer 11 unicode(1+65) as Letter gibt B usw.

Ist die Ziffer kleiner als 10, dann können wir sie vorn anfügen.

Von unserer Dezimalzahl ziehen wir die Ziffer ab (das war ja der Rest bei Division durch 16) und teilen die Differenz durch 16. Damit erhalten wir ein ganzzahliges Ergebnis.

```
hexToDezi( hex )
script variables potenz dezi i
set dezi to 0
set potenz to 16 ^ (length of text hex − 1)
for i = 1 to length of text hex
    if letter i of hex > 9
        change dezi by (unicode of letter i of hex − 55) × potenz
    else
        change dezi by letter i of hex × potenz
    set potenz to potenz / 16
report dezi
```

Abbildung 5.9.: Hexadezimal nach Dezimal

Bei der Umrechnung von Hexadezimal- in Dualzahlen orientieren wir uns an dem Algorithmus aus Kapitel 6.3. Das Zahlensystem hat jetzt allerdings die Basis 16, so dass wir bei der Berechnung der Potenz als Basis 16 statt 2 einsetzen müssen und die Potenz durch 16 statt durch 2 teilen. Bei den Ziffern müssen wir zwischen den Ziffern von 0 bis 9 und den Hexadezimalziffern von A bis F unterscheiden. Bei den Ziffern von 0 bis 9 können wir wie in 6.3 das Produkt aus der aktuellen Ziffer und der aktuellen Potenz zum Ergebnis addieren.

Bei den Ziffern von A bis F müssen wir den Unicode der Ziffer bilden. Diese liegen zwischen 65 und 70. Wir ziehen also 55 ab und erhalten damit den Faktor, mit dem wir die Potenz multiplizieren müssen.

```
+hexBin(+ data +,+ mode +)+
script variables hList bList i
set hList to list 0 1 2 3 4 5 6 7 8 9 A B C D E F
set bList to list 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
for i = 1 to 16
  if mode = hex
    if data = item i of hList
      report item i of bList
  else
    if data = item i of bList
      report item i of hList
report Fehler!
```

Abbildung 5.10.: Umwandlung Hexadezimal Dual

Einfacher gestaltet sich die Umrechnung von Hexadezimalzahlen in Dualzahlen und umgekehrt. Nach der Konstruktion der Hexadezimalzahlen kann jede hexadezimale Ziffer durch eine vierstellige Dualzahl dargestellt werden und umgekehrt. Wir implementieren deshalb einen Block, der eine einzelne Hexadezimalziffer in einer vierstellige Dualzahl umrechnet bzw. eine vierstellige Dualzahl in eine Hexadezimalziffer. Der erste Parameter gibt die Ziffer bzw. die vier Ziffern an, der zweite die Richtung (hex für Umwandlung von Hexadezimalzahlen, bin für die Umwandlung von Dualzahlen).

```
+hexToBin(+ data +)+
script variables result i
set result to
for i = 1 to length of text data
  set result to join result hexBin( letter i of data , hex )
report result
```

Abbildung 5.11.: Hexadezimal nach Dual

Vor der Umrechnung müssen wir die Dualzahlen mit führenden Nullen so lange auffüllen, bis die Gesamtlänge eine durch Vier teilbare Zahl ist. Dann gehen wir die Daten zeichenweise bzw. in Blöcken von vier Zeichen durch und wandeln sie um.

```
+binToHex(+ data +)+
script variables i bin result
set result to
repeat until length of text data mod 4 = 0
  set data to join 0 data
for i = 0 to length of text data / 4 − 1
  set bin to join letter i × 4 + 1 of data
                  letter i × 4 + 2 of data
                  letter i × 4 + 3 of data
                  letter i × 4 + 4 of data
  set result to join result hexBin( bin , bin )
report result
```

Abbildung 5.12.: Dual nach Hexadezimal

## 5.7. Aufgaben

**Aufgabe 5.1** *Wandle um in eine Dualzahl. Dokumentiere dabei die Rechnung durch eine Trace-Tabelle:*

- *109*
- *81*

**Aufgabe 5.2** *Wandle um in eine Dezimalzahl. Dokumentiere dabei die Rechnung durch eine Trace-Tabelle:*

- *1010111*
- *1100100*

**Aufgabe 5.3** *Addiere bzw. subtrahiere durch Addition des Zweierkomplements schriftlich. Kontrolliere das Ergebnis durch Umwandlung der Zahlen und des Ergebnisses in Dezimalzahlen.*

- *101010 + 110110*
- *1100100 + 1101101*

- *110110 - 101010*

- *1101101 - 1100100*

**Aufgabe 5.4** *Implementieren Sie den Wechselgeld-Automaten zunächst in der fehlerhaften Form, testen Sie dessen Funktionsweise und verbessern Sie den Automaten dann, indem Sie alle Dezimalzahlen sofort nach der Division auf zwei Nachkommastellen runden.*

**Aufgabe 5.5** *Die folgende Aufgabe befasst sich mit dem Umrechnen zwischen verschiedenen Zahlsystemen.*

- *Konstruiere einen Dual-Dezimal-Konvertierer, der eine beliebige Eingabe in das Dualsystem bzw. in das Dezimalsystem umwandelt. Dabei sollten die Dualzahl ein vorangestelltes kleines* ***b*** *haben. Die Dezimalzahlen werden ohne Kennzeichnung aus- bzw. eingegeben.*

- *Ergänze den Konvertierer zum Dual-Dezimal-Hexadezimal-Konvertierer. Die Hexadezimalzahlen werden dabei ein vorangestelltes kleines* ***h*** *gekennzeichnet. Beispiel: Die Eingabe hF6 gibt aus: b11110110, 246 und hF6.*

# 6. Umgang mit Zeichenketten

In diesem Kapitel lernen Sie

- wie man Buchstaben mit dem ASCII- bzw. Unicode codiert,
- welche Zeichenkettenoperationen Snap! standardmäßig zur Verfügung stellt,
- ie man neue Zeichenkettenoperationen implementiert und
- wie man Algorithmen unter Verwendung elementarer Zeichenkettenoperationen implementiert.

## 6.1. ASCII-Code

Die Abkürzung ASCII steht für **American Standard Code for Information Interchange**, zu deutsch Amerikanischer Standardcode für den Informationsaustausch. Um Buchstaben und damit Texte zwischen Computern und Druckern austauschen zu können, benötigt man eine Möglichkeit, diese Buchstaben in Dualzahlen zu übersetzen. Mit dem ASCII, den die American Standards Association (ASA) 1963 billigte, versuchte man einen einheitlichen Standard für eine solche Übersetzung zu schaffen. Die ASA wählte als Basis eine siebenstellige Dualzahl. Damit gibt es 128 verschiedene mögliche Kombinationen. Davon wurden 33 für nicht druckbare Zeichen reserviert und 95 für Buchstaben und Satzzeichen. Unter den nicht druckbaren Zeichen sind z.B. die 7 (000 0111), die für *Bell* steht, also ein Läuten in der Fernschreibmaschine auslöst. Schreibmaschinen arbeiten mit Hebeln oder einem Kugelkopf, die die Buchstaben immer an die gleiche Stelle drucken. Um einen fortlaufenden Text zu schreiben, wird die Walze mit dem Papier nach jedem Anschlag ein kleines Stück weiter gedreht. Zu Beginn einer neuen Zeile erfolgt ein Carriage Return (13 = 000 1101, d.h. die Walze geht nach links bis zum Anschlag, und ein Line Feed (10 = 000 1010), d.h. die Walze wird eine Zeile weitergedreht. Bis heute stehen deshalb Carriage Return und Line Feed für einen Zeilenumbruch.

Die Großbuchstaben finden sich zwischen 65 (A) und 90 (Z), die Kleinbuchstaben zwischen 97 (a) und 122 (z). Die deutschen Umlaute und das ß sind nicht in der ursprünglichen Codierung enthalten. Es gibt deshalb verschiedene Erweiterungen, die den ursprünglichen ASCII-Zeichensatz um weitere Zeichen ergänzen. Diese sind aber nicht einheitlich. Betrachtet man z.B. deutschsprachige Text, die unter DOS erstellt wurden, unter Windows, so werden die Umlaute falsch dargestellt, weil DOS und Windows unterschiedliche Erweiterungen des ursprünglichen 7-Bit-ASCII-Zeichensatzes verwenden.

Aus dem Bestreben, alle Sprachen weltweit einheitlich zu codieren, entstand der Unicode. Er stellt eine Erweiterung des ASCII-Codes dar. Langfristig soll er jedem Schriftzeichen in allen Sprachen eine digitale Darstellung zuweisen. Mit zunächst 16 Bits waren 65 536 verschiedene Kombinationen möglich. Neue Fassungen verwenden bis zu 32 Bits. Die deutschen Umlaute werden mit 196 (Ä), 214 (Ö), 220 (Ü), 223 (ß), 228 (ä), 246 (ö) und 252 (ü) codiert.

## 6.2. Buchstaben

In Snap! haben wir zwei Funktionen, die der Umwandlung von einzelnen Buchstaben, englisch **Char** genannt, in Unicode bzw. umgekehrt dienen. Es handelt sich bei den beiden Funktionen um:

unicode of a

unicode 65 as letter

Abbildung 6.1.: Unicode

Beide Funktionen arbeiten mit dem Unicode. Die Operation `unicode of` liefert die Dezimalzahl des jeweiligem Zeichens. Man übergibt einen Buchstaben als Parameter und erhält einen Wert. Der Befehl `unicode ... as letter` gibt das Zeichen an, das zu einer gegebenen Dezimalzahl passt. Wenn man die Befehle ineinander schachtelt `unicode of (unicode zahl as letter)` dann gibt Snap! die ursprüngliche Zahl aus.

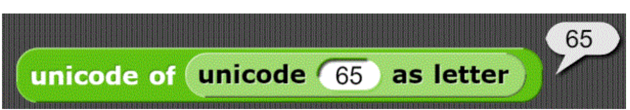



Abbildung 6.2.: Unicode Beispiel 1

In diesem Beispiel gibt Snap! 65 als Lösung an.

Wenn man mehrere Buchstaben als Parameter eingibt, z.B. `unicode of (hallo)`, dann gibt Snap! eine Liste der Unicodes der Buchstaben aus.

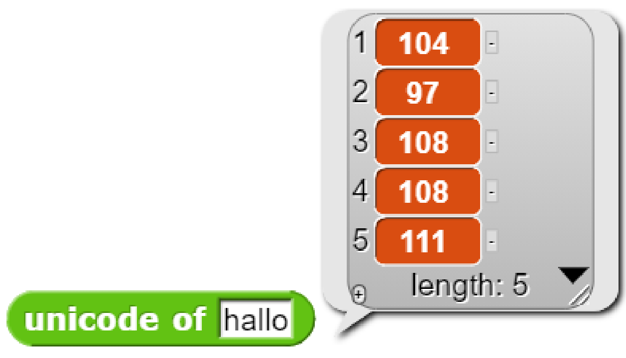



Abbildung 6.3.: Unicode Beispiel 2

## 6.3. Zeichenketten

Zeichenketten werden im Englischen als **String** bezeichnet. Die Ergänzenden Hinweise zum Kerncurriculum Informatik geben für die Arbeit mit Zeichenketten im Rahmen von zentralen Prüfungsaufgaben die folgenden sechs Operationen vor:

- Bestimmen der Länge einer Zeichenkette, `length of wort`
- Auslesen eines Zeichens an einer bestimmten Position, `letter i of wort`

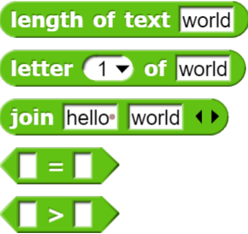



Abbildung 6.4.: Operationen mit Zeichenketten

- Ersetzen einer Zeichens an einer bestimmten Position
- Verbinden von zwei Zeichenketten zu einer, `join( wort1, wort2 )`
- Prüfen des Inhalts von zwei Zeichenketten auf Gleichheit, der `=`-Operator
- Lexikographisches Vergleichen von zwei Zeichenketten

Vier dieser Operationen sind in Snap! bereits enthalten. Lediglich das Ersetzen eines Zeichens an einer bestimmten Position und eine Operator für die lexikographische Ordnung müssen wir noch programmieren.

Eine Zeichenkette kann mit dem `Set`-Befehl in einer Variable gespeichert werden. Auch hier gilt, dass die Verwendung des `Change`-Befehls die Zeichenkette löscht und auf Null setzt. Zu den Operationen im Einzelnen:

Die Länge einer Zeichenkette kann mit dem grünen `length of text wort`-Block ausgegeben werden (hier: 5). Dieser grüne Block darf nicht mit dem braunen `length of`-Block verwechselt werden. Der grüne Block gibt die Länge einer Zeichenkette aus, der braune die Länge einer Liste. Zwar gibt in Snap! der braune Block eine Fehlermeldung aus, wenn er keinen Parameter vom Typ Liste erhält (**Error expecting list but getting Boolean / Number / Text**). Umgekehrt gibt der grüne Block die Liste aus, wenn er eine Liste als Parameter erhält. Um die Verwechslungsgefahr zu verringern, wurde mit der Version 6.0 von Snap! der grüne Block in `length of text` umbenannt. Der `length of text`-Block ist ein Reporter und kann deshalb nicht allein in einer Zeile stehen.

`letter i of wort` gibt den i-ten Buchstaben einer Zeichenkette aus. Wird ein Wert außerhalb der Reichweite des gegebenen Strings als Parameter angegeben, so liefert der Block keinen Wert. Voreingestellte Werte für den ersten Parameter sind `1` (der erste Buchstabe), `random` (ein zufälliger Buchstabe) und `last` (der letzte Buchstabe). Auch `letter i of wort` kann als Reporter nicht allein in einer Zeile stehen.

Mit `join` können mehrere Zeichenketten aneinander gesetzt werden. Mit den kleinen schwarzen Pfeilen lässt sich die Anzahl beliebig erhöhen oder verringern. Beachten Sie, dass auch der `join`-Block ein Reporter ist, obwohl er ein Verb als Namen trägt, und deshalb **nicht für sich allein in einer Quelltextzeile** stehen kann.

Die Gleichheit von zwei Zeichenketten lässt sich mit dem `=`-Block überprüfen. Dabei gibt es eine Besonderheit: Groß- und Kleinbuchstaben werden als gleich angesehen, z.B. `Auf=auf`. Das gilt auch für die Umlaute: `Ä=ä`. Benötigt man einen Block, der bei Zeichenketten zwischen Groß- und Kleinschreibung unterscheidet, so muss man die einzelnen Buchstaben durch ihren Unicode ersetzen und die entstehenden Zahlen vergleichen.

Der <- und der >-Reporter unterscheiden im Unterschied zu =-Block zwischen Groß- und Kleinbuchstaben. Das hat die Folge, dass Snap! **beide folgende Gleichungen als `true` bewertet**: Ä=ä und Ä<ä. Generell sind alle Großbuchstaben kleiner als die Kleinbuchstaben, weil ihr ASCII-Code kleiner ist.

## 6.4. Grundrezept Operationen mit Zeichenketten

Snap! verfügt standardmäßig nicht über Operationen, um Buchstaben in einer Zeichenkette zu ändern. Mit `join` lassen sich nur Buchstaben vorn oder hinten anfügen. Immer dann, wenn ein Buchstabe in einem String eingefügt, gelöscht oder verändert werden soll, müssen wir das Wort buchstabenweise neu aufbauen. Alle diese Aufgaben folgen einem Grundmuster:

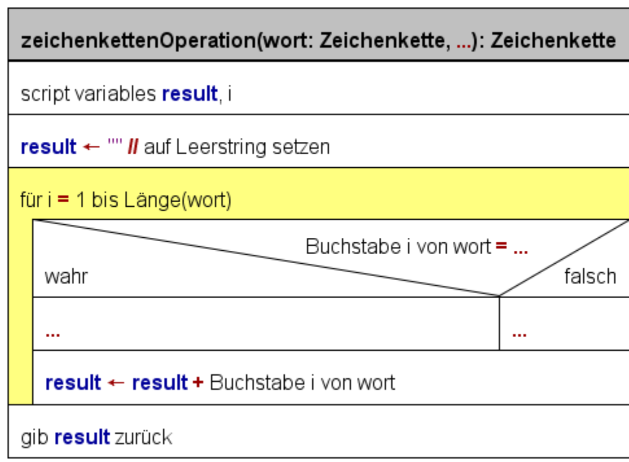



Abbildung 6.5.: Grundrezept Operation mit Zeichenketten

Wir benötigen zwei lokale Variable, eine Zählvariable und eine Variable für das Ergebnis, häufig als `result` bezeichnet. `Result` wird zunächst auf den Leerstring gesetzt, um die automatisch eingetragene Null zu löschen. Dann gehen wir mit einer `FOR`-Schleife die gesamte Zeichenkette durch. Wir prüfen, ob die Buchstabe an der aktuellen Stelle (oder diese Stelle) eine Bedingung erfüllt. In Abhängigkeit von diesem Vergleich werden dann die Buchstaben an das Ergebnis angehängt. Nach Durchlaufen der Schleife wird das Ergebnis zurückgegeben.

Als Beispiel für einen solchen Block betrachten wir die Operation `ersetzeAn(Letter,Stelle,Wort)`, die in einem als Parameter gegebenen `Wort` an der angegebenen `Stelle` den Buchstaben durch `Letter` ersetzt. Diese Operation kann im Abitur vorausgesetzt werden (vgl. Ergänzende Hinweise zum Kerncurriculum Informatik, I.).

Wir benötigen zwei Skriptvariable, `result` für das Ergebnis und `i` als Zählvariable. Zu Beginn löschen wir die Variable `result`, damit wir keine überflüssige Null am Anfang haben. Dann gehen wir mit einer `FOR`-Schleife das `wort` vom ersten bis zum letzten Buchstaben durch. Wenn die Zählvariable `i` den gleichen Wert wie `stelle` hat, fügen wir `letter` hinten an, sonst den Buchstaben, der bisher an der i-ten Stelle stand. Am Schluss wird `result` ausgegeben.

Um eine Zeichenkette auf dem Bildschirm auszugeben, steht in der Palette **Pen** der Befehl `write Text size Punkte` zur Verfügung. Der Befehl schreibt den angegebenen Text an die

Abbildung 6.6.: `ersetzeAn(letter, stelle, wort)`

aktuelle Position in die aktuelle Richtung.

## 6.5. Lexikographie

Die Identität von Groß- und Kleinbuchstaben ist beschränkt auf die Gleichheitsoperation. Der >- bzw. <-Block ordnen die Buchstaben nach dem Unicode an. Es gilt also **A < Z < a < z < Ä < ß < ä**. Bei der lexikographischen Sortierung werden Worte wie in einem Lexikon sortiert. D.h., Groß- und Kleinbuchstaben werden nicht unterschieden und die Umlaute werden aufgelöst bzw. ersetzt: **A** < **aa** < **AAb** < **Aachen** < **...** < **aha** < **Ähre** < **Akne** < **...** < **haschen** < **Häschen** < **hat** < **...** < **Masse** < **Maßeinheit** < **...** < **Zu** < **zz**. Satzzeichen werden durch ein Leerzeichen ersetzt. Die näheren Einzelheiten für die lexikographische Sortierung in der deutschen Sprache sind in der DIN 5007 geregelt. Dort sind zwei Varianten für den Umgang mit Umlauten festgelegt. Bei der ersten Variante werden die Umlaute aufgelöst, d.h. ä wird durch ae ersetzt usw. Diese Variante wird in Lexika verwendet. Für Namenslisten sieht die DIN 5007 eine zweite Variante vor: Dabei wird ä durch a, ö durch o und ü durch u ersetzt. In beiden Varianten wird ß durch ss ersetzt.

Für den lexikographischen Vergleich zweier Zeichenketten müssen wir also folgende Vorarbeiten leisten:

- Auflösen der Umlaute
- Umwandlung der Klein- in Großbuchstaben
- Entfernen von Leer- und Satzzeichen

Auf die so modifizierten Zeichenketten können wir die >- bzw. <-Operatoren anwenden. Damit die Operation übersichtlich bleibt, lagern wir das Auflösen der Umlaute aus.

Abbildung 6.7.: Umlaute ersetzen

Dazu erstellen wir einen Reporter `Umlaut` mit einem Parameter `letter`, der vier einseitige Verzweigungen enthält. Da der Gleichheitsoperator nicht zwischen Groß- und Kleinschreibung unterscheidet, sind alle Möglichkeiten abgedeckt. Die Auflösung wird in Großbuchstaben zurückgegeben. Am Schluss fügen wir ein `report letter` an, um alle anderen Zeichen weiterzugeben. Das Beispiel zeigt die Auflösung der Umlaute für Lexika. Für Namenslisten, wo ä durch a usw. ersetzt wird, muss der Block entsprechend angepasst werden. Bei Bedarf können auch Regeln für das Umwandeln von Akzenten eingefügt werden.

Für den lexikographischen Vergleich erstellen wir einen eigenen Block mit dem Parameter `wort`. Im ersten Schritt wandeln wir die Umlaute um, dann werden die Klein- in Großbuchstaben umgewandelt und die Großbuchstaben angehängt, wobei wir Satzzeichen und Leerzeichen weglassen. Deshalb verwenden wir neben `result` und der Zählvariable `i` eine Variable `zw`(ischen). Nach dem Löschen von `zw` gehen wir das Wort zeichenweise durch, wobei wir eine `FOR`-Schleife verwenden. Dabei wird das Ergebnis schrittweise zusammengesetzt, wobei

```
+lexi(+ wort +)+
script variables result i zw
set zw to
for i = 1 to length of text wort
    set zw to join zw umlaut( letter i of wort )
set result to
for i = 1 to length of text zw
    if letter i of zw ≥ A and letter i of zw ≤ Z
        set result to join result letter i of zw
    if letter i of zw ≥ a and letter i of zw ≤ z
        set result to join result unicode unicode of letter i of zw − 32 as letter
report result
```

Abbildung 6.8.: Lexikographischer Vergleich

wir mit dem Block die `Umlaute` die Umlaute auflösen.

Mit der zweiten `FOR`-Schleife fügen wir zunächst die Großbuchstaben an. Diese liegen zwischen haben A=65 und Z=90 (jeweils einschließlich). Die Kleinbuchstaben liegen zwischen a=97 und z=122. Um einen Kleinbuchstaben in den entsprechenden Großbuchstaben umzuwandeln, ermitteln wir den zugehörigen Unicode, ziehen 32 ab (die Differenz von 97 und 65) und wandeln das Ergebnis wieder in einen Buchstaben um. Alle übrigen Buchstaben und Satzzeichen fallen durch unser Raster und werden nicht ans Ergebnis angefügt.

## 6.6. Galgenraten

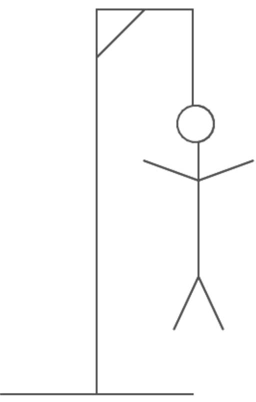


Abbildung 6.9.: Galgen

Galgenraten ist ein Ratespiel für zwei Spieler, das mit einem Blatt Papier und einem Stift gespielt werden kann. Spieler A denkt sich ein Geheimwort aus. Er teilt Spieler B nur mit, aus wie viel Buchstaben das Geheimwort besteht. Spieler B schreibt oben auf das Blatt Papier so viele Striche, wie Buchstaben in dem Wort enthalten sind. Dann denkt er sich einen Buchstaben aus und fragt Spieler A, ob dieser in dem Geheimwort enthalten ist. Wenn der Buchstabe enthalten ist, dann muss ihm Spieler A sagen, an welchen Stellen dieser Buchstabe im Geheimwort vorkommt. Spieler B trägt diese Buchstaben in seinem Schema ein. Wenn der Buchstabe nicht enthalten ist, hat Spieler B einen Fehler gemacht und er muss den nächsten Strich oder den nächsten Kreis für seinen Galgen zeichnen. Er gewinnt, wenn er das Geheimwort erraten kann, bevor der Galgen fertig ist. Das Bild zeigt einen Galgen aus 11 Teilen:

Unser `ersetzeAn`-Reporter aus Kapitel 6.4 muss für Galgenraten modifiziert werden. Wir müssen nicht mehr einen Buchstaben an einer bestimmten Stelle ersetzen, sondern an allen Stellen, an denen er im Geheimwort vorkommt. An allen anderen Stellen steht der bisherige Buchstabe.

```
+ersetze+(+ buchstabe +,+ wort +)+
script variables result i
set result to
for i = 1 to length of text wort
    if buchstabe = letter i of geheimwort
        set result to join result letter i of geheimwort
    else
        set result to join result letter i of wort
report result
```

Abbildung 6.10.: Ersetze für Galgenraten

Welche globalen Variablen benötigen wir?

1. Wir wollen nicht immer mit dem gleichen Geheimwort spielen, sondern mit unterschiedlichen. Also benötigen wir eine globale Variable `geheimwort`.

2. Während des Spiels verändert sich unser

Wort mit jedem Buchstaben, den wir richtig erraten. Wir benötigen also eine Variable `wort`.

3. Wir müssen die Anzahl der Fehler mitzählen. Dafür benötigen wir eine Variable `fehler`.

Für alle weiteren Variablen sollten wir Parameter oder Skriptvariable verwenden.

Das Spiel besteht aus drei Phasen: Vorbereitung, die eigentlichen Spielzüge und am Schluss die Auswertung (gewonnen oder verloren).

1. Vorbereitung: Wir wählen ein zufälliges `Geheimwort` aus, erzeugen ein `Wort` der gleichen Länge aus Unterstrichen und setzen die `Fehler` auf Null.

   Später werden an dieser Stelle auch die Vorbereitungen für das Zeichnen getroffen.

```
set Fehler to 0
set geheimwort to item random of list Esel Pferd Kuh
set Wort to
for i = 1 to length of text geheimwort
    set Wort to join Wort _
```

Abbildung 6.11.: Start Galgenraten

2. Spielzüge: Die Spielzüge werden so lange wiederholt, bis die Variable `Fehler` den Wert 11 erreicht, oder `Wort=Geheimwort` ist. Innerhalb dieser Wiederholungsschleife wird zunächst gefragt, welcher Buchstabe geraten werden soll. Ist `Wort=ersetze( letter 1 of answer, word`, so wird die Zahl der Fehler um 1 erhöht (An dieser Stelle wird später auch das nächste Stück des Galgens gezeichnet). Falls nicht, wird `wort` durch `ersetze(letter 1 of answer, wort)` ersetzt.

3. Abschluss: Falls `(Wort = Geheimwort)` Siegesfanfare ... sonst leider verloren, du musst noch üben!

   Hinweise zum Zeichnen des Galgens finden sich in Kapitel 3.4.

```
repeat until Fehler > 11 or Wort = geheimwort
    ask Welchen Buchstaben will Du raten? and wait
    if Wort = ersetze ( letter 1 of answer , Wort )
        change Fehler by 1
    else
        set Wort to ersetze ( letter , Wort )
```

Abbildung 6.12.: Spielzüge

## 6.7. Zeichenketten mit Listenbefehlen

Während Snap! sparsam mit Befehlen für Zeichenketten ausgestattet ist, gibt es für Listen viel mehr Möglichkeiten, z.B. einen Befehl, um ein Listenelement an einer bestimmten Stelle in einer Liste zu ändern. Wir können also unsere Zeichenkettenoperationen auch als Listenoperationen programmieren, wenn wir die Zeichenkette in eine Liste umwandeln und das Ergebnis wieder in eine Zeichenkette.

Zum Glück stellt uns Snap! dafür zwei Blöcke zur Verfügung, die diesen Vorgang vereinfachen. Der `split word by letter`-Block teilt eine Zeichenkette in Listenelemente auf, wobei der zweite Parameter angibt, nach welchen Kriterien aufgeteilt werden soll. `By letter` schreibt jeden Buchstaben in ein Listenelement, `by word` jedes Wort. Für das Umkehren der Operation können wir einfach den Befehl `join` mit einem Parameter benutzen. Unser Block zur Ersetzung eines Buchstabens an einer bestimmten Stelle eines Wortes vereinfacht, sich also:

```
+ersetzeAn(+ letter +,+ stelle # +,+ wort +)+
script variables result
set result to split wort by letter
replace item stelle of result with letter
report join result
```

Abbildung 6.13.: Zeichenkettenoperation mit Listenbefehlen

Blöcke zum Einfügen (mit `REPLACE`) oder Löschen (mit `DELETE`) lassen sich entsprechend konstruieren.

## 6.8. Aufgaben

**Aufgabe 6.1** *Implementiere eine Operation* `delete(wort: Zeichenkette, stelle, anzahl: Ganzzahl): Zeichenkette`*: Die Operation* `delete` *soll aus dem* `wort` *ab* `stelle` `anzahl` *Buchstaben löschen, z.B.* `delete(informatik,3,5)` *liefert* `intik`*.*

*Teste die Lösung u.a. mit den Beispielen*

- `delete(Mathematik,1,2)`*,*
- `delete(Mathematik,8,3)` *und*
- `delete(Mathematik,8,3)`*.*

**Aufgabe 6.2** *Implementiere eine Operation* `insert(wort, teilwort: Zeichenkette, stelle: Ganzzahl): Zeichenkette`*: Die Operation* `insert` *soll in das* `wort` *ab* `stelle` *das* `teilwort` *einfügen, z.B.* `insert(Matik,thema,3)` *liefert* `Mathematik`*.*

**Aufgabe 6.3** *Gesucht ist ein Block* `beforeLastletter(wort: Zeichenkette): Buchstabe`*, der den vorletzten Buchstaben eines Wortes liefert.*

**Aufgabe 6.4** *Gesucht ist ein Block* `quersumme(zahl: Ganzzahl): Ganzzahl`*, der die Quersumme einer Zahl liefert.*

**Aufgabe 6.5** *Gesucht ist ein Block* `paritybit(dual: Zeichenkette): Zeichenkette`*, der zu einer beliebigen Dualzahl (die als Zeichenkette vorliegt) ein Paritätsbit so berechnet, dass die Anzahl der Einsen zusammen mit dem Paritätsbit gerade ist, und der die Zahl mit dem angehängten Paritätsbit ausgibt.*

*Bei der Eingabe von 0110 liefert ParityBit 01100, bei der Eingabe 111011 liefert ParityBit 1110111.*

**Aufgabe 6.6** *Gesucht ist ein Block* `reverse(wort: Zeichenkette): Zeichenkette`*, der ein Wort von hinten nach vorn gelesen ausgibt.*

**Aufgabe 6.7** *Gesucht ist ein Block* `palindromTest(wort): Wahrheitswert`*, der prüft, ob ein Wort ein Palindrom ist. Ein Palindrom ist eine Zeichenkette, die von vorne und hinten gelesen das gleiche Wort ergibt, z.B. Rentner. Dabei wird nicht zwischen Groß- und Kleinschreibung unterschieden.*

**Aufgabe 6.8** *Implementiere Operationen zum Löschen eines Buchstabens, zum Ersetzen eines Buchstabens und zum Einfügen einer Zeichenkette in ein Wort mit Hilfe von Listenbefehlen.*

**Aufgabe 6.9** *Implementieren Sie Galgenraten. Das Spiel soll durch Klick auf die grüne Flagge starten und außerdem eine Erläuterung des Spielprinzips und der Benutzersteuerung enthalten.*

# 7. Sortieren

In diesem Kapitel lernen Sie

- wie man Daten in Listen speichert und auf diese Daten zugreift und
- wie man Listen sortiert.

## 7.1. Listen

Eine der wichtigsten Alltagserfahrungen ist, das man Dinge leichter wiederfindet, wenn man sie geordnet abgelegt hat. Dafür muss man sie sortieren. Entsprechend sind Sortierverfahren ein wichtiger Bereich der Informatik. Bevor wir Dinge sortieren, benötigen wir Datenstrukturen, die mehrere Exemplare einer Zahl o.ä. fassen. Dafür verwenden wir Listen.

Eine Reihung (ARRAY) ist ein Datentyp, für den ein fester Bereich im Speicher reserviert und in gleich große Teilstücke aufgeteilt wird. Snap! kennt den Datentyp Reihung nicht, wir verwenden statt dessen dynamische Liste, deren Größe automatisch an den Inhalt angepasst wird.

Listen können in Snap! erstellt werden, indem man eine Variable über **Make a variable** oder eine Scriptvariable erstellt und dieser durch den Befehl `set myList to list` {} eine Liste zuweist.

Der Reporter `list` liefert eine neue Liste. Über die kleinen schwarzen Pfeile kann eingestellt werden, wie viele Elemente die Liste haben haben soll. Es ist möglich, eine leere Liste zu erzeugen oder bei Entstehung gleich Werte in die Liste einzutragen. Handschriftlich notieren wir dies wie folgt: Soll die Liste leer sein, markieren wir dies durch die beiden nebeneinander stehenden geschweiften Klammern. Enthält die Liste bereits bei ihrer Erstellung Elemente, werden diese zwischen den geschweiften Klammern aufgezählt.

Der Reporter `item 1 of Liste` liefert ein Element der Liste. Voreingestellt für den ersten Parameter sind `1`, `last` und `random`. Dabei bedeutet `random` ein zufälliges Element der Liste. Für den ersten Parameter können auch Variable eingesetzt werden.

Die Länge einer Liste, also die Anzahl an Elementen, lässt sich in Snap! durch den braunen Befehl `length of Liste` bestimmen. Der grüne Befehl `length of text Zeichenkette` dient zur Bestimmung der Länge einer Zeichenkette, also beispielsweise zur Längenbestimmung eines Elements der Liste. Snap! gibt bei falscher Benutzung des braunen Befehls ein **Error expecting list but getting text** aus.

Der Commandblock `add thing to Liste` fügt ein neues Element hinten an die gegebene Liste an. **Achtung, häufiger Anfängerfehler:** Der Befehl add ist in Snap! nicht zur Addition von Zahlen geeignet! Dafür ist das Plus-Zeichen reserviert bzw. das `change Variable by Wert`.

Abbildung 7.1.: Listenbefehle

Der Commandblock `delete 1 of Liste` löscht das angegebene Element der Liste. Die folgenden Elemente rücken automatisch entsprechend auf. Voreingestellt für den ersten Parameter sind `1`, `last` und `all`. Für den ersten Parameter können auch Variablen eingesetzt werden. Wird ein Wert als Parameter angegeben, die grösser ist als die Länge der Liste, bleibt der Block wirkungslos.

Der Commandblock `insert thing at 1 of Liste` fügt an der angegebenen Stelle ein neues Listenelement ein. Voreingestellt für den zweiten Parameter sind `1`, `last` und `random`. Dabei bedeutet `random` ein zufälliges Element der Liste. Für den zweiten Parameter können auch Variablen eingesetzt werden. Wird eine Zahl als Parameter angegeben, die um 1 größer ist als die Länge der Liste, wird das neue Element hinten angefügt. Ist die Zahl noch größer, bleibt der Block wirkungslos.

Der Commandblock `replace item 1 of Liste with thing` ersetzt das Listenelement an der angegebenen Stelle mit dem als drittem Parameter angegebenen neuen Listenelement. Voreingestellt für den ersten Parameter sind `1`, `last` und `random`. Dabei bedeutet `random` ein zufälliges Element der Liste. Für den ersten Parameter können auch Variablen eingesetzt werden. Wird eine Zahl als Parameter angegeben, die größer ist als die Länge der Liste, bleibt der Block wirkungslos.

Der Prädikatblock `Liste contains thing` prüft, ob das als zweiter Parameter angegebene Element in der Liste enthalten ist.

Neu ist der Prädikatblock `is MyList empty?`, der prüft, ob die als Parameter übergebene Liste leer ist. In älteren Struktogrammen oder Programmen findet sich deshalb die äquivalente Bedingung `length of MyList = 0`.

In Struktogrammen werden Listenelemente häufig mit dem Namen der Liste und einem nachgestellten Index in eckigen Klammern gekennzeichnet:

| ZugriffAufLlistenelemente |
|---|
| minimum ← Matrix [zeile, spalte] |
| Liste[i] ← Zufalllszahl von 1 bis 100 |

Abbildung 7.2.: Zugriff auf Listenelemente in Struktogrammen

In der ersten Zeile wird der Variable `minimum` das Element zugewiesen, das unter „Zeile/Spalte“ in der Matrix steht: `set minimum to item spalte of (item zeile of Matrix)`. In der zweiten Zeile wird dem i-ten Element der Liste eine Zufallszahl zwischen 1 und 100 zugewiesen: `replace item i of Liste with pick random 1 to 100`.

Schließlich spielen zwei Befehle aus dem grünen Operators-Menü eine wichtige Rolle beim Umgang mit Listen: Mit dem `split ... by` Block lässt sich eine Zeichenkette in eine Liste überführen. Dabei gibt es unterschiedliche Versionen. Mit `split Zeichenkette by letter` erhält man eine Liste, die jeden einzelnen Buchstaben der Zeichenkette als Element enthält. Mit `split Zeichenkette by word` erhält man eine Liste, die jedes Wort der Zeichenkette als Element enthält. Dabei ist versteht Snap! unter einem Wort eine Folge von Zeichen, die durch ein oder mehrere Leerzeichen getrennt sind. Mit `join (Liste)` erstellt man aus einer Liste, egal ob sie aus Einzelbuchstaben oder aus Worten besteht, wieder eine einzige Zeichenkette. Dabei wird der `JOIN`-Block mit dem schwarzen Links-Pfeil auf einen Parameter eingestellt.

## 7.2. Grundrezept InsertionSort

InsertionSort bzw. Sortieren durch Einfügen kann man mit einem Kartenspiel wie folgt umsetzen: Auf dem Tisch liegt ein Stapel Karten. Man zieht nacheinander die oberste Karte und nimmt diese so in die Hand, dass die Karten von links nach rechts der Größe nach geordnet sind. Dabei muss man drei Fälle unterscheiden:

- die Hand enthält noch keine Karten. Dann nimmt man die erste Karte in die Hand.

- die Hand enthält nur Karten, die kleiner sind als die neu gezogene. Dann nimmt man die neue Karte als letzte auf.
- die Hand enthält einige Karten, die größer sind als die neu gezogene. Dann wird die neue Karte vor der ersten einsortiert, die größer ist.

Für Sortieren durch Einfügen ergibt sich folgendes Struktogramm (siehe Abbildung 6.2):

| **insertionSort (data: Liste): Liste** |
| --- |
| script variables **i**, **result** |
| **result** ← leere Liste |
| wiederhole bis (data ist leer) |
|     i ← 1 |
|     wiederhole bis (data[1] < **result**[i] or i > Länge(**result**)) |
|         i ← i+1 |
|     füge data[1] an der Stelle i in **result** ein |
|     lösche data[1] |
| gib **result** zurück |

Abbildung 7.3.: InsertionSort

Andere Programmiersprachen benötigen für die drei verschiedenen Fälle drei verschiedene Befehle: einen für das Einfügen in einer leere Liste, einen für das Anfügen hinten an eine Liste und einen für das Einfügen zwischen zwei Elementen. Der Commandblock `insert thing at 1 of Liste` deckt alle drei Fälle ab. Wenn die Hand leer ist, ist beim ersten Element i=1 bereits größer als die Länge der Liste. Dann wird an der ersten Stelle eingefügt. Findet das Programm eine Karte, die größer ist als die neue, dann kann man mit `insert` die Karte an dieser Stelle einfügen. Der Block funktioniert nämlich, wie in 8.1. erläutert, auch an der ersten Stelle hinter der Liste.

Ein zweiter Hinweis an dieser Stelle: Beim Sortieren durch Einfügen wird immer die erste Karte von dem Stapel auf dem Tisch gezogen. Das Verfahren eignet sich also nicht nur für Listen, sondern auch für solche Datentypen, bei denen man nur auf das erste Element zugreifen kann. Wir werden später einige solcher Datentypen kennenlernen, z.B. den Stapel (STACK) und die Schlange (QUEUE). Unser Block zum Sortieren könnte also wie folgt aussehen:

```
+insertionSort(+ data : +)+
script variables result i
set result to list
repeat until length of data = 0
    set i to 1
    repeat until item 1 of data < item i of result or i > length of result
        change i by 1
    insert item 1 of data at i of result
    delete 1 of data
report result
```

Abbildung 7.4.: InsertionSort

Unser Block zum Sortieren ist ein Reporter, der als Parameter die Daten in Form einer Liste erhält - wir nennen sie `data` -, die sortiert werden sollen. Wir verwenden als Skriptvariablen `result` für das Ergebnis und `i` als Zählvariable. Zunächst erstellen wir eine leere Liste für `result`. Dann folgt eine Schleife, die so lange wiederholt wird, bis die Länge unserer Liste `data` gleich Null ist, also bis alle Elemente sortiert wurden. Nun müssen wir in jedem Durchgang die Stelle finden, an der wir einfügen müssen. Diese Stelle kann am Ende der Ergebnisliste sein, aber auch dort, wo wir das erste Element finden, das größer ist als das einzusortierende. Wir können also keine `FOR`-Schleife mit einem festen Endwert verwenden, sondern müssen unsere Zählschleife händisch konstruieren.

Wir beginnen mit der Initialisierung, `i` erhält den Wert Eins. Dann folgt eine Wiederholung, bis eine von zwei Bedingungen erfüllt ist. Entweder ist `i` größer als die Länge von `result` oder das erste Element von `data` ist kleiner als das aktuelle Element der Ergebnisliste. Ist die Abbruchbedingung nicht erfüllt, erhöhen wir den Wert der Zählvariable `i` um Eins. Wenn wir die Schleife verlassen, enthält die Variable `i` die Stelle, an der wir das neue Element einfügen müssen. Wir fügen das Element dort

ein und löschen es aus `data`. Sind alle Elemente einsortiert, wird `result` zurückgegeben und der Block damit beendet.

## 7.3. Grundrezept SelectionSort

SelectionSort kann man wie folgt veranschaulichen: Man nimmt die Karten unsortiert auf die Hand und wiederholt dann folgenden Vorgang, bis die Hand leer ist: Man geht die Karten von links nach rechts durch und sucht die jeweils kleinste Karte heraus. Diese legt man dann ab auf einen Stapel. Ist die Hand mit den Karten leer, so findet man auf dem Stapel die Listen sortiert vor. Diesem Auswahlvorgang (englisch: Selection) verdankt das Sortierverfahren seinen Namen.

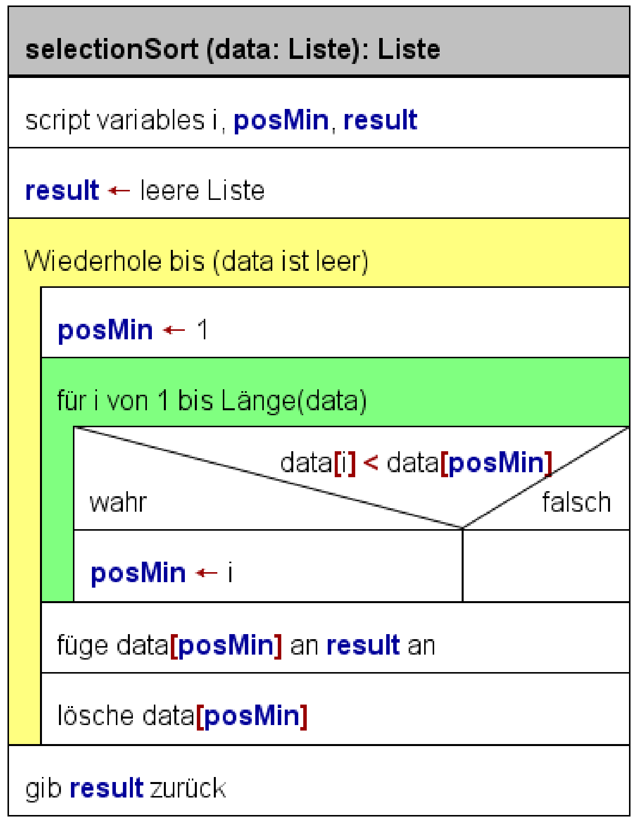



Abbildung 7.5.: Selektionsort

Wir benötigen drei Skriptvariable: `result` für das Ergebnis, eine Zählvariable  und eine Variable `posMin`, in der wir die Position des Minimums speichern. Wir setzen `result` auf den gewünschten Zieltyp und beginnen mit der äußeren Wiederholungsschleife, die wir so lange durchlaufen, bis unsere `Data`-Liste leer ist. Innerhalb der äußeren Schleife setzen wir zunächst `posMin` auf Eins. Dann gehen wir alle Elemente von `Data` mit einer FOR-Schleife durch. Wenn wir ein Element finden, dass kleiner ist als das Element an `posMin`, dann speichern wir dessen Position in der Variable `posMin`. Nach dem Ende der inneren Schleife wird das kleinste Element in das Ergebnis angefügt und aus `Data` gelöscht.

```
+selectionSort(+ data : +)+
script variables i posMin result
set result to list
repeat until length of data = 0
    set posMin to 1
    for i = 1 to length of data
        if item i of data < item posMin of data
            set posMin to i
    add item posMin of data to result
    delete posMin of data
report result
```

Abbildung 7.6.: SelectionSort-Block

Das Verfahren lässt sich einfach implementieren, wobei wir hier voraussetzen, dass das Ergebnis eine Liste ist. Die Anpassung für Stapel, Schlange und dynamische Reihung behandeln wir in Jahrgang 12.

## 7.4. Bubblesort

Das BubbleSort-Verfahren beruht darauf, dass jeweils zwei benachbarte Elemente miteinander vergleichen werden und gegebenenfalls getauscht werden. Im Unterschied zu den bisher behandelten Sortierverfahren wird das BubbleSort-Verfahren in-place durchgeführt, d.h. innerhalb der bestehenden Liste. Das BubbleSort-Verfahren benötigt eine Hilfsvariable zum Tauschen, weil man den Inhalt zweier Speicherstellen nicht unmittelbar tauschen kann, sondern den einen Wert in einem Zwischenspeicher festhalten muss.

In der Grundform werden jeweils zwei benachbarte Elemente miteinander verglichen und ggf. getauscht, beginnend beim ersten bis zu vorletzten. Dieser Schritt wird n mal wiederholt, wobei n die Anzahl der Elemente ist.

**bubbleSort(data: Liste):Liste**

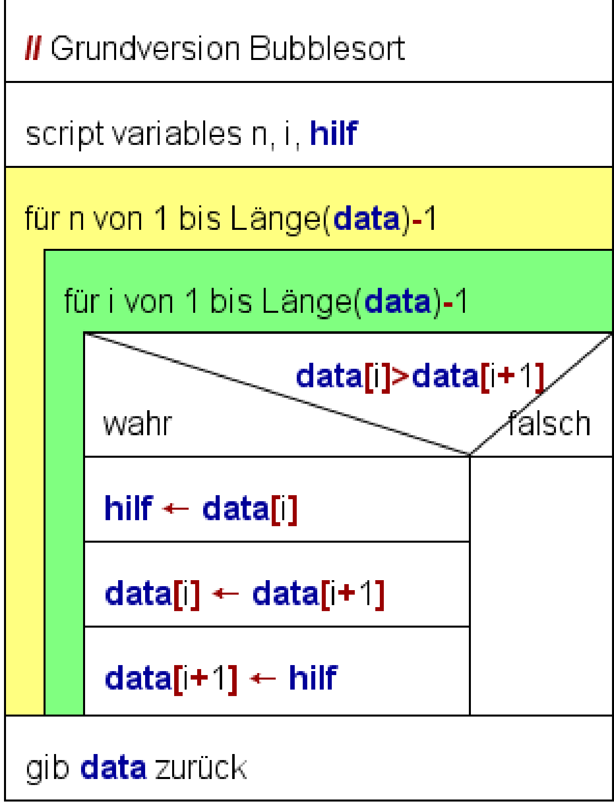



Abbildung 7.7.: BubbleSort-Grundform

Nach dem ersten Durchgang befindet sich das größte Element an der letzten Stelle, nach dem zweiten befindet sich das zweitletzte Element an der zweitletzten Stelle usw. In der ersten Variation wird deshalb die Anzahl der Elemente, die verglichen werden, bei jedem Durchgang ums Eins verringert.

Eine Liste ist vollständig sortiert, wenn bei einem Durchgang kein Tausch mehr durchgeführt wurde. Diese Eigenschaft kann man dazu nutzen, um das BubbleSort-Verfahren weiter zu optimieren.

Dazu führen wir eine Variable `sortiert` ein, die in der äußeren Schleife abgefragt wird. Diese Variable wird beim Start des Programms auf `false` gesetzt, damit wir überhaupt in die Wiederholungsschleife `repeat until sortiert` hineinkommen. Zu Beginn jedes Durchgangs wird `sortiert` auf `true` gesetzt. Entdecken wir dann beim Durchgang durch die Liste ein Element, das größer ist als das folgende, dann führen wir einen Tauschvorgang durch und setzen `sortiert` auf `false`, weil wir ja noch nicht fertig sind. Erst wenn wir einen kompletten Durchgang von ersten bis zum vorletzten Element ohne Tauschen absolviert haben, ist die Liste sortiert und wir verlassen die `repeat`-Schleife und damit den Block.

```
+bubbleSort(+ data +)+
script variables i sortiert hilf
set sortiert to false
repeat until sortiert
    set sortiert to true
    for i = 1 to length of data - 1
        if item i of data > item i + 1 of data
            set hilf to item i of data
            replace item i of data with item i + 1 of data
            replace item i + 1 of data with hilf
            set sortiert to false
report data
```

Abbildung 7.8.: optimiertes BubbleSort

Die optimierte Form des BubbleSort ist besonders gut dafür geeignet, teilsortierte bzw. vorsortierte Listen zu sortieren. Im Unterschied zu InsertionSort und SelectionSort bricht der Algorithmus nämlich ab, sobald kein Tauschvorgang mehr vorgenommen werden muss. Hat man z.B. eine sortierte Liste und fügt ein neues Element vorn hinzu, dann gelangt dieses Element bereits beim ersten Durchgang an seinen Platz und beim zweiten Durchgang signalisiert die Hilfsvariable `gefunden`, dass die Liste wieder sortiert ist.

## 7.5. Aufgaben

**Aufgabe 7.1** *Implementiere eine Operation `insertionSort(data: Liste): Liste`, die eine Liste als Parameter erhält, und eine sortierte Liste ausgibt.*

**Aufgabe 7.2** *Implementiere eine Operation* `selectionSort(data: Liste): Liste`*, die eine Liste als Parameter erhält, und eine sortierte Liste ausgibt.*

**Aufgabe 7.3** *Implementiere eine Operation* `bubbleSort(data: Liste)`*, die eine Liste als Parameter erhält und diese Liste sortiert.*

**Aufgabe 7.4** *Lesen Sie getrennt vier Zeichen ein. Lassen Sie im Anschluss ausgeben, ob die vier Zeichen alphabetisch aufsteigend sortiert eingegeben wurden. Dabei gilt die ASCII-Reihenfolge, d.h. Großbuchstaben stehen vor den Kleinbuchstaben. Sollte ein Zeichen der Eingabe kein Buchstabe gewesen sein, so quittieren Sie dies mit einer Fehlermeldung.*

**Aufgabe 7.5** *Erstellen Sie ein Programm, welches ein Zeichen einliest. Lassen Sie im Anschluss ausgeben, ob es sich bei der eingegebenen Zahl um ein Sonderzeichen, einen Kleinbuchstaben, einen Großbuchstaben oder um eine Ganzzahl handelt.*

*Hinweis: Im Anhang finden Sie einen Ausschnitt der ASCII-Code-Tabelle.*

**Aufgabe 7.6** *Beschreiben Sie eine Operation zur Bestimmung der größten Zahl aus einer Liste von Zahlen mit Hilfe eines Struktogramms.*

# 8. Verschlüsselungsverfahren

In diesem Kapitel lernen Sie

- was der Unterschied zwischen einer Geheimschrift und einem Kryptosystem ist,
- wie die Prinzipien der Transposition und der Substitution zur Verschlüsselung von Daten funktioniert,
- wie man ein monoalphabetisches Verfahren implementiert am Beispiel des Caesar-Verfahrens,
- wie man ein Transpositionsverfahren implementiert am Beispiel von Skytale,
- wie das Prinzip der Häufigkeitsanalyse funktioniert und
- wie sicher einfache Verschlüsselungsverfahren sind.

## 8.1. Kryptosysteme

Nachrichten vor der Übermittlung zu verschlüsseln, um zu verhindern, dass Dritte mitlesen können, und gegnerische Nachrichten zu entschlüsseln, ist seit langem eine Herausforderung. Die Erfindung des Computers hat sowohl die Möglichkeiten der Kryptographie (= Verschlüsselung von Texten) als auch der Kryptoanalyse (= Entschlüsselung von Texten) enorm ausgeweitet. Kryptoverfahren sind deshalb ein wichtiger Bestandteil des Informatikunterrichtes.

Jede Schrift basiert auf einer endlichen Menge von Zeichen, die Alphabet genannt wird. Die Zeichen des Alphabets nennen wir Symbole oder Buchstaben. Texte sind Folgen von Symbolen (Buchstaben).

Unsere heutige Schrift hat sich über verschiedene Stufen entwickelt: Zunächst entstanden Bildschriften wie die ägyptischen Hieroglyphen. Später folgten Wortschriften wie das Chinesisch. Es entstanden Silbenschriften wie die

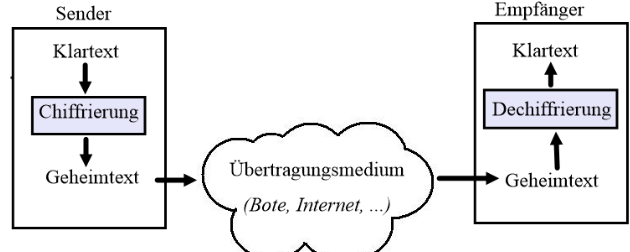



Abbildung 8.1.: Kommunikation mit einer Geheimschrift

Linearschrift B, die Schrift des mykenischen Griechenland, die sich von Knossos auf Kreta aus verbreitete. Heute benutzen wir im europäischen Kulturkreis überwiegend Buchstabenschriften.

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | $A$ | $B$ | $\Gamma$ | $\Delta$ | $E$ |
| 2 | $Z$ | $H$ | $\Theta$ | $I$ | $K$ |
| 3 | $\Lambda$ | $M$ | $N$ | $\Xi$ | $O$ |
| 4 | $\Pi$ | $P$ | $\Sigma$ | $T$ | $\Upsilon$ |
| 5 | $\Phi$ | $X$ | $\Psi$ | $\Omega$ | |

Abbildung 8.2.: Tabelle für Geheimschrift Polybios

Mit der Verbreitung der Schrift entwickelte sich auch das Bedürfnis, geschriebene Texte vor unbefugtem Lesen zu schützen. Es entstanden die ersten Geheimschriften. Eine der ältesten dokumentierten Geheimschriften entwickelte der griechische Schriftsteller Polybios (ca. 200 v.Chr.). Er schrieb die 24 Buchstaben des griechischen Alphabets in ein rechteckiges Schema und ordnete dann jedem Buchstaben eine zweistellige Zahl zu, wobei die erste für die Zeile und die zweite für die Spalte steht. Somit steht 14 für den griechischen Buchstaben $\Delta$ (sprich: Delta).

Eine Geheimschrift besteht aus vier Bestandteilen: einem Klartextalphabet, einem Geheimalphabet, einer Vorschrift zur Chiffrierung und einer Vorschrift zu Dechiffrierung. Am Bei-

spiel der Geheimschrift Polybios lauten die Bestandteile;

**Klartextalphabet:** Greek (griechisches Alphabet)

**Geheimalphabet:** {1, 2, 3, 4, 5}

**Chiffrierung:** Lies den Klartext von links nach rechts ein und ersetze jeden Buchstaben durch die Folge von zwei Ziffern aus dem Geheimalphabet. Die erste Ziffer ist die Nummer der Zeile, in der sich der Buchstabe befindet. Die zweite Ziffer ist die Nummer der Spalte, in der sich der Buchstaben befindet.

**Dechiffrierung:** Unterteile den Geheimtext in Zahlenpaare und suche zu jeden Zahlenpaar den zugehörigen Buchstaben heraus.

Wichtig dabei ist, dass nicht nur die einzelnen Buchstaben wieder eindeutig zurück übersetzt werden können, sondern auch ganze Texte. Ein System, dass jedem Buchstaben seine Platznummer im Alphabet zuordnet, also A=1, B=2, ..., K=11, L=12, ..., Z=26, ist keine Geheimschrift, weil der Geheimtext 12 sowohl als AB als auch als L übersetzt werden könnte.

Das Problem der Geheimschriften ist, dass sie wenig Vielfalt bieten. Ist das Verfahren einmal bekannt, so kann jeder Gegner den Geheimtext entziffern. Gewünscht ist deshalb eine größere Vielfalt der Codierungen. Diese Vielfalt bieten Kryptosysteme, die zusätzlich zu Klartextalphabet, Geheimalphabet und Vorschriften zu Chiffrierung und Dechiffrierung Schlüssel verwenden.

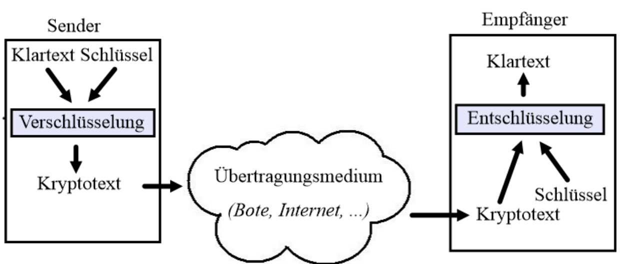



Abbildung 8.3.: Kommunikation mit einem Kryptosystem

Auguste Kerckhoffs (1835-1903) war ein niederländischer Linguist und Kryptologe. Er veröffentlichte 1883 seine Schrift *La Cryptographie militaire.*

Das Werk beinhaltet Kerckhoffs' Prinzip für die „strategische Kryptologie“, nach damaligem Verständnis die vertrauliche Kommunikation über Telegraphenleitungen. Er formulierte: **Ein Kryptosystem ist sicher, wenn sich, trotz öffentlich bekanntem Verschlüsselungsverfahren, die ursprünglichen Klartexte nicht ohne Kenntnis des Schlüssels aus den Geheimtexten ableiten lassen**.

Dem Kerckhoff'schen Prinzip wird oft die sogenannte „Security by Obscurity“gegenübergestellt: Sicherheit durch Geheimhaltung des (Verschlüsselungs-)Algorithmus, möglicherweise zusätzlich zur Geheimhaltung des Schlüssels.

Kerckhoffs stellte weitere Regeln und Anforderungen für vertrauliche Kommunikation auf:

- der Schlüssel muss leicht zu merken und auswechselbar sein,
- die Geheimtexte müssen übertragbar sein, das heißt damals telegraphierbar,
- der Chiffrierapparat und die Dokumente müssen transportierbar sein,
- das System muss einfach (ohne Expertenhilfe) zu benutzen sein.

Bei **symmetrischen Verfahren**, auf die wir uns in der Einführungsphase beschränken, gibt es jeweils einen Schlüssel, der zum Ver- und Entschlüsseln genutzt wird (Entsprechend verfügen asymmetrische Verfahren jeweils über zwei Schlüssel, einen öffentlichen und einen privaten).

Die symmetrischen Verfahren werden wiederum in zwei Hauptgruppen unterteilt: Bei **Substitutionsverfahren** werden die Buchstaben durch andere Buchstaben oder Zeichenkombinationen ersetzt. Als Beispiel für ein klassisches Substitutionsverfahren behandeln wir das Caesar-Verfahren. Bei **Transpositionsverfahren** bleiben die Buchstaben erhalten, lediglich ihre Positionen innerhalb des Textes werden vertauscht. Als Beispiel behandeln wir Skytale. Mittels eine Häufigkeitsanalyse lassen sich beide Verfahren unterscheiden und auch entschlüsseln.

## 8.2. Caesar

Eines der ältesten Kryptoverfahren wurde von dem römischen Feldherrn und Politiker Julius Caesar verwendet und ist deshalb nach ihm benannt. Die Funktionsweise des Caesar-Verfahrens kann man sich am besten mit der Caesar-Scheibe klarmachen. Sie besteht aus zwei Kreisen unterschiedlicher Größe, die am Rand jeweils mit einem Alphabet beschriftet sind. Man wählt einen Schlüsselbuchstaben und dreht die innere Scheibe so, dass das äußere A über dem inneren Schlüsselbuchstaben zu stehen kommt. Dann kann man für jeden Buchstaben des Klaralphabets auf der äußeren Scheibe den zugehörigen Buchstaben des Geheimalphabets auf der inneren Scheibe ablesen. Julius Caesar benutzte übrigens eine vereinfachte Version: Er verwendete grundsätzlich den Schlüssel D.

Für die beiden Kreise der Caesar-Scheibe verwenden wir zwei globale Variable, nämlich `Alphabet` und `Geheimalphabet`. Unter diesen Namen speichern wir jeweils eine Liste mit dem (deutschen) Alphabet in Großbuchstaben. Dafür gibt es verschiedene Möglichkeiten: Entweder vergrößern wir die Liste, so dass sie 26 Elemente umfasst, und tragen das Alphabet ein. Oder wir benutzen eine `FOR`-Schleife, um die Buchstaben zu den Unicodes von 65=A bis 90=Z in die Liste einzufügen:

Abbildung 8.4.: Erzeugung eines Alphabets

Nachdem wir Alphabet und Geheimalphabet erzeugt haben, speichern wir in einer weiteren globalen Variable `key` den Schlüssel. Nun müssen wir die innere Scheibe „drehen“. Dazu fügen wir den ersten Buchstaben hinten an und löschen ihn vorn (bitte genau in dieser Reihenfolge!). Das Ganze wird so lange wiederholt, bis der erste Buchstabe des Geheimalphabets mit dem `key` übereinstimmt.

Für das Heraussuchen des Geheimbuchstabens zu einem gegebenen Buchstaben nutzen wir den Block `index of Element in Liste`. Er liefert den Index, d.h. die Stelle, an das beschriebene Element zum ersten Mal in der Liste steht. Wir nutzen diesen Block, um die Stelle zu bestimmen, an der der aktuelle Buchstabe in der Liste Alphabet steht. Dann fügen wir den Buchstaben an das Ergebnis an, der an dieser Stelle im Geheimalphabet steht.

Abbildung 8.5.: Scheibe drehen

Abbildung 8.6.: Buchstaben verschlüsseln

Diese Zeile leistet drei Dinge:

1. Sie sucht zu jedem Buchstaben den zugehörigen Geheimbuchstaben heraus.
2. Sie wandelt dabei Kleinbuchstaben in Großbuchstaben um, solange im Geheimalphabet Großbuchstaben stehen.
3. Sie filtert den Text, indem sie Leerzeichen, Satzzeichen und Ziffern löscht, weil der `index of Element in Liste`-Block nur dann einen Wert zurückgibt, der von Null verschieden ist, wenn das Element ein Buchstabe ist.

Damit können wir unseren Block `caesar(text, key)` fertigstellen:

Für das fertige Programm benötigen wir noch eine Möglichkeit der Eingabe des Schlüssels und kurzer Texte. Längere Texte importiert man am besten über das Kontextmenü unten rechts an der Liste.

Wir wollen aber Texte, die mit dem Caesar-Verfahren verschlüsselt wurden, auch wieder entschlüsseln. Dazu stellen wir einige Vorüberlegungen an: Die Verschlüsselung mit A verschiebt die Buchstaben nicht (lediglich

Abbildung 8.7.: Caesar-Verfahren

die Kleinbuchstaben werden in Großbuchstaben umgewandelt und Ziffern, Satzzeichen und Leerzeichen werden entfernt). Die Verschlüsselung mit B verschiebt alle Buchstaben um eine Stelle. Wenn wir die Buchstaben um 25 Stellen, also mit Z, verschieben, erhalten wir wieder den ursprünglichen Text. Die Verschlüsselung mit C verschiebt alle Buchstaben um zwei Stellen. Zur Entschlüsselung müssen sie um 24 Stellen verschoben werden, also mit Y.

Abbildung 8.8.: Entschlüsselung des Caesar-Verfahrens

Die Anzahl der Stellen, um die ein Buchstabe verschiebt, erhalten wir, indem wir von seinem Unicode 65 abziehen: Unicode von A = 65, 65-65=0. Unicode von B = 66, 66-65=1, Unicode von C = 67; 67-65=2, ..., Unicode von Z = 90, 90-65=25. Verschlüsselungs- und Entschlüsselungsbuchstabe müssen zusammen alle Buchstaben um 26 Stellen verschieben. Eine vorläufige Lösung lautet also: Ziehe vom Unicode des Key 65 ab, ziehe diese Differenz von 26 ab, addiere 26 dazu und wandelt das Ergebnis in einen Buchstaben um. Die Formel funktioniert in in meisten Fällen, aber es gibt eine Ausnahme und zwar A. Unicode von A = 65. 65-65=0. 26-0=26. 26+65=91. 91 in ein Symbol umgewandelt ergibt [. Das Problem entsteht, weil wir im vorletzten Schritt 65 zu 26 addiert haben und damit den Bereich der Großbuchstaben verlassen. Wir müssen also das Ergebnis der zweiten Differenz mod 26 nehmen: 26 mod 26 = 0, 0+65=65, Symbol zu 65 ist A. Die Formel heißt also richtig: `((26 - (Unicode von Key - 65) mod 26) + 65`, wobei die Ergebniszahl dann mit `unicode ... as letter` in einen Großbuchstaben umgewandelt werden muss.

## 8.3. Skytale

„Ene mene mu und raus bist Du.“ Mit diesem Abzählreim wird jeder 7. aus einer Runde heraus gewählt, bis nur noch einer übrig ist. Man zählt damit 1,2,3,4,5,6,7,1,2,3,4,5,6,7,1,2,...

In der Informatik zählen wir häufig bis zur einer bestimmten Zahl und fangen dann wieder bei 1 an. Dazu verwenden wir den MOD-Befehl aus Kapitel 5. Mit seiner Hilfe können wir unser „Ene Mene Mu“ umsetzen. Wir verwenden eine Zählvariable und setzen sie auf 1. In jedem Schritt prüfen wir, ob das Ergebnis schon größer ist als die obere Schranke s (hier: 7), ziehen ggf. s ab und addieren dann Eins hinzu. Das könnte man mit einer Fallunterscheidung machen oder durch i mod s.

Abbildung 8.9.: Zählen von 1 bis n

Eines der ältesten bekannten Kryptosytem wurde ca. 500 v.Chr. von den Spartanern entwickelt und verwendet. Sie wickelten einen schmalen Pergamentstreifen spiralförmig um einen Stab und schrieben darauf ihre Nachricht. Der Empfänger wickelte den Streifen um einen Stab gleichen Durchmessers und konnte so den Klartext lesen.

Der Schlüssel ist die Anzahl der Seiten, die unser Stab hat, hier also sechs. Außer dem Schlüssel benötigen wir noch eine andere Information, nämlich wie lange unsere Zeilen sein

Abbildung 8.10.: Skytale-Lederstreifen

**skytale(text: Zeichenkette; key: Zahl): Zeichenkette**

| Erhöhe die Textlänge auf das nächste Vielfache von key |
| --- |
| Berechne die Anzahl der Windungen (loops) |
| Erstelle eine Liste stab und füge loops leere Zeichenketten an |
| Verteile die Buchstaben des Klartextes auf die Windungen |
| Gib die Zeichenketten des Stabs nacheinander aus |

Abbildung 8.11.: Skytale-Prinzip

müssen, genauer, wie oft wir den Streifen um den Stab wickeln müssen. Dabei gilt:

**Anzahl der Stabseiten mal Anzahl der Windungen = Länge des Klartextes**

Bei Schlüssel s=6 schreiben wir die einzelnen Zeichen des Textes fortlaufend abwechselnd in (Textlänge durch 6)-Windungen und lesen den Geheimtext dann zeilenweise ab.

Für den Stab verwenden wir eine einfache Liste, für jede einzelne Windung eine Zeichenkette. Die einzelnen Buchstaben werden an die zugehörige Windung hinten angehängt. Am Schluss werden alle Windungen aneinander gehängt und ergeben so den Kyrptotext.

Beispielnachricht: WIRSCHREIBENEINEKLAUSURAMSECHZEHHNTENOKTOBER Wir verwenden als Skytale einen Holzstab mit dem Querschnitt eines regelmäßigen Sechsecks und erzeugen Zeilen mit acht Buchstaben, also acht Windungen.
WIKMNB
IBLSTE
REAEER
SNUCN
CESHO
HIUZK
RNRET
EEAHO
Das ergibt den Geheimtext WIKMNBIBLSTEREAEERSNUCN CESHO HIUZK RNRET EEAHO

Man sieht sofort an den Leerzeichen, dass die Schlüssellänge sechs ist. Wenn wir die Leerzeichen aber einfach weglassen, können wir nicht mehr richtig entschlüsseln. Unsere Nachricht hat 43 Buchstaben. Um eine durch 6 teilbare Zahl zu erhalten, ergänzen wir 5 zufällige Buchstaben (hier XXXXX zu Übungszwecken, sonst ebenso ungeeignet wie die Leerzeichen!).

Wir brauchen 48:6=8 Windungen, da wir 48 Buchstaben auf 6 Zeilen verteilen müssen. WIRSCHREIBENEINEKLAUSURAMSECHSZEHNTENOKTOBERXXXXX ergibt
WIKMNB
IBLSTE
REAEER
SNUCNX
CESHOX
HIUZKX
RNRETX
EEAHOX
und als Geheimtext WIKMNBIBLSTEREAEERSNUCNXCESHOXHIUZKXRNRETXEEAHOX.

Wenn wir Skytale implementieren, erstellen wir einen Reporter mit den Parametern Text und Key, wobei der Schlüssel hier eine Zahl ist. Um den Block fehlerfrei zu erstellen geht man am besten schrittweise vor. Die Grafik 2.7 zeigt die grundlegenden Schritte, die man abarbeiten muss.

Es bietet sich an, nach jedem Schritt den Reporter entsprechend anzupassen. Z.B. könnte man nach dem ersten Schritt ein `report text` einfügen, um zu prüfen, die die Buchstaben richtig angefügt wurden. Nach dem dritten Schritt lässt man sich mit `report stab` anzeigen, ob der Stab richtig erstellt und mit der erforderlichen Anzahl von Windungen ausgestattet wurde. Ist ein Schritt erreicht, so schiebt man den Reporter nach unten und implementiert den nächsten Schritt.

Durch 6 teilbar sind alle Zahlen, die bei Divi-

sion durch 6 den Rest 0 ergeben: `zahl mod 6 = 0`. Wenn wir das Alphabet als globale Variable in Form einer Liste definiert haben, dann erhalten wir einen zufälligen Buchstaben mit `item random of alphabet`.

**skytale(text: Zeichenkette;key: Zahl):Zeichenkette**

script varibles i, **j**, **loops**, **result**, **stab**

wiederhole bis (Länge(text) **mod** key)=0

  zufälligen Buchstaben zu text hinzufügen

**loops** ← Länge(text)/key

**stab** ← leere Liste

für i von 1 bis **loops**

  füge "" zu **stab** hinzu

**j** ← 1

für i von 1 bis Länge(text)

  ersetze **stab[j]** durch join(**stab[j]**, letter i of text)

  **j** ← (**j mod loops**)+1

**result** ← ""

für i von 1 bis **loops**

  füge **stab[i]** an **result** an

gib **result** zurück

Abbildung 8.12.: Skytale

Wir gehen den Text mit zwei Zählvariablen durch: i läuft von 1 bis zur Länge des Textes und gibt den Buchstaben an, der gerade angefügt wird, j läuft von 1 bis Anzahl der Windungen (`loops`) und gibt die Windung an, in die der aktuelle Buchstabe eingefügt wird.

Zum Entschlüsseln teilen wir unseren Geheimtext auf 6 Windungen auf: WIKMNBIBLSTEREAEERSNUCNXCESHO-XHIUZKXRNRETXEEAHOX
WIRSCHRE
IBENEINE
KLAUSURA
MSECHZEH
NTENOKTO
BERXXXXX
WIRSCHREIBENEINEKLAUSURAMSECH-ZEHNTENOKTOBERXXXXX

+decryptSkytale+(+ text +,+ key +)+
report skytale ( text , length of text text / key )

Abbildung 8.13.: Entschlüsseln von Skytale

Jede der sechs Windungen enthält jetzt acht Buchstaben (statt acht Teile mit je sechs Buchstaben). Wir können also Skytale entschlüsseln, indem wir den Schüssel (key) und die Anzahl der Windungen (loops) vertauschen. An dem Beispiel sieht man, warum es sinnvoll ist, die Kryptoverfahren als Reporter mit Parametern zu vereinbaren: so kann man sie einfach zusammensetzen.

## 8.4. Häufigkeit

**haeufigkeit(text: Zeichenkette):Liste**

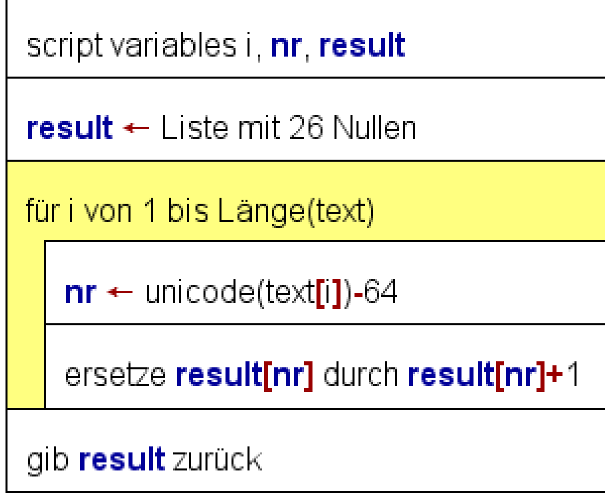



Abbildung 8.14.: Häufigkeit Struktogramm

Der erste Schritt, eine Verschlüsselung zu knacken, ist das Auszählen der Buchstaben. Dazu erzeugen wir eine Liste mit 26 Nullen und gehen dann den Text durch. Für jeden Buchstaben erhöhen wir den entsprechenden Eintrag in der Liste um Eins. Die Häufigkeit des Buchstabens A kommt in das erste Element der Liste, die Häufigkeit von B in das

zweite Element usw. bis zur Häufigkeit von Z, die wir im 26. Element der Liste speichern. Jetzt benötigen wir nur noch eine Formel, um jedem Buchstaben seine Platznummer zuzuordnen. Dazu nehmen wir den Unicode eines Buchstabens und ziehen 64 ab. Der Unicode von A ist 65. Davon 64 abgezogen bleibt 1, die Ordnungszahl von A.

Unser Block `haeufigkeit` funktioniert nur mit Großbuchstaben, weil wir keinen Vergleich mit dem Gleichheitszeichen durchführen (das würde ja Groß- und Kleinbuchstaben gleich behandeln), sondern auf den Unicode zurückgreifen. Falls der Text noch Satzzeichen, Leerzeichen oder Kleinbuchstaben enthält, kann man ihn mit `caesar(text, A)` umwandeln.

Abbildung 8.15.: Implementierung Häufigkeit

## 8.5. Balkendiagramm

Wenn wir die Häufigkeit der Buchstaben eines Kryptotextes bestimmt haben, wollen wir sie als Balkendiagramm darstellen. Ein solches Balkendiagramm ist leichter überschaubar als 26 Tabellenwerte. Insgesamt sind noch drei Schritte erforderlich: Die x-Achse muss beschriftet werden, wir müssen die Grundlinie für die Balken festlegen und dann die Balken in der richtigen Höhe zeichnen.

Bei der Beschriftung der x-Achse müssen wir beachten, dass es prinzipiell zwei verschiedene Arten von Schriften gibt. Wir verwenden hier in diesem Skript Times Roman, die zu den proportionalen Schriften gehört. Bei proportionalen Schriften richtet sich die Breite eines Buchstabens nach dem jeweiligen Symbol. So ist ein i schmaler als ein o oder gar ein m. Zeitschriften und Bücher werden ganz überwiegend in proportionalen Schriftarten gedruckt. Dagegen sind bei nichtproportionalen Schriften alle Buchstaben gleich breit. Für die Beschriftung unseres Balkendiagramms ist das von Vorteil, weil wir so alle Balken gleich breit machen können.

Snap! hat ab der Version 5 unter **Pen** einen Block `write` zur Textausgabe auf dem Bildschirm, der eine unproportionale Schrift verwendet:

Abbildung 8.16.: Textausgabe auf dem Bildschirm

Auf welcher y-Koordinate sollen die Säulen beginnen? Unser Standard-Bildschirmausschnitt unter Snap! ist 480 Pixel breit und 360 Pixel hoch. Der Ursprung (0—0) ist in der Bildschirmmitte. Unsere y-Koordinaten laufen also von -100 bis +180. Weil wir meist unterschiedliche Häufigkeiten vergleichen wollen, z.B. Kryptotext und deutsche Sprache, müssen wir die Balkendiagramme in unterschiedlicher Höhe zeichnen können. Wir wollen aber nicht zwei verschiedene Blöcke implementieren. Deshalb verwenden wir für die untere y-Koordinate einen Parameter namens `yOffset`. Im Beispiel ist yOffset = -120.

Wir haben längere und kürzere Texte, deren Häufigkeit wir darstellen wollen. Also benötigen wir zum Zeichnen die relative Häufigkeit. Der häufigste Buchstabe in der deutschen Sprache ist das „E“, das einen Anteil von etwas unter 20% hat. Also soll 20% die höchste Säule sein. Wenn wir 130 Pixel für die Säulen vorsehen, dann entspricht 100%=1 650 Pixeln. Die relative Häufigkeit entspricht dann x Pixeln und es gilt:

$$\frac{x}{650} = \frac{relativeHäufigkeit}{1}$$

$$x = relativeHäufigkeit \cdot 650$$

```
+summe(+ myList ⋮ +)+
script variables result i ◂▸
set result ▾ to 0
for i = 1 to length of myList
    change result ▾ by item i of myList
report result
```

Abbildung 8.17.: Berechnung der Summe

Die relative Häufigkeit erhalten wir, indem wir die absolute Häufigkeit eines Buchstabens durch die Summe aller Buchstaben teilen.

Man könnte natürlich statt 20% der Gesamtsumme auch jeweils den größten Wert mit 130 Pixeln darstellen. Es gibt aber Kryptoverfahren, die zu einer gleichmäßigeren Verteilung der Buchstaben führen. Diesen Effekt sieht man nicht mehr, wenn immer der größte Wert eine Säule von 130 Pixeln Höhe hat.

**barChart(list: Liste; yOffset: Zahl)**

```
script variables i, sum
sum ← summe(list)
pen up
pen width ← 5
für i von 1 bis 26
    gehe zu y: 17*i-220 y: yOffset
    pen down
    gehe zu x: 17*i-220 y: yOffset+650*list[i]/sum
    pen up
```

Abbildung 8.18.: Barchart-Struktogramm

Die FOR-Schleife läuft von 1 bis 26, weil wir 26 Balken zeichnen wollen. Wenn wir oben unter Tools **Flat line ends** anklicken, erhalten wir rechteckige Balken (sonst werden die Enden abgerundet). Die 17 steht für die Säulenbreite, gemessen jeweils von Anfang bis Anfang.

```
+barChart(+ list ⋮ +,+ yOffset # +)+
script variables i sum ◂▸
set sum ▾ to summe( list )
set pen size to 5
pen up
for i = 1 to 26
    go to x: -220 + i × 17 y: yOffset
    pen down
    go to x: -220 + i × 17 y: yOffset + 650 × item i of list / sum
    pen up
```

Abbildung 8.19.: Barchart

Eine einzelne Häufigkeitsverteilung sagt nicht viel aus, wir müssen sie vergleichen, z.B. mit der Häufigkeitsverteilung der deutschen Sprache. Die folgende Grafik zeigt die Häufigkeitsverteilung eines längeren Gedichts, „Die Glocke“ von Friedrich Schiller:

Abbildung 8.20.: Häufigkeitsverteilung der deutschen Sprache

Man sieht deutlich, dass das „E“ mit 17,4% hervorragt. Aber ein weiteres charakteristisches Muster zeichnet sich aber. Wenn wir von E aus 4 Schritte weit gehen, dann landen wir auf dem I. Nach weiteren 5 Schritten landen wir auf dem N und dann nach vier Schritten auf dem R. An das R schließt eine Art Treppe an. Dieses 4-5-4-Muster ist charakteristisch für die deutsche Sprache und auch charakteristisch für die Caesar-Verschlüsselung.

Am Beispiel erkennt man: Der häufigste Buchstabe ist jetzt das P. Geht man jetzt im 4-5-4-Muster weiter, so landet man auf

Abbildung 8.21.: Häufigkeitsverteilung eines mit CAESAR verschlüsselten Kryptotextes

T, Y und C, alles relativ hohe Balken. Außerdem schließt sich an C die Treppe an. Es liegt also tatsächlich eine Verschiebung und damit eine Caesar-Verschlüsselung vor. Den Schlüsselbuchstaben erhalten wir, indem wir vom höchsten Balken aus vier Buchstaben zurückgehen, weil wir von E bis A ja auch vier Buchstaben zurückgehen müssen. Hier wurde mit L verschlüsselt.

Mit Hilfe der Häufigkeitsanalyse können wir also unterscheiden, ob ein Transpositionsverfahren vorliegt (dann ergibt sich eine Häufigkeitsverteilung ähnlich der deutschen Sprache) oder ob ein Substitutionsverfahren vorliegt (dann ergibt sich eine abweichende Häufigkeitsverteilung wie im Beispiel oben).

## 8.6. Aufgaben

**Aufgabe 8.1** *Implementiere eine Operation* `caesar(data: Zeichenkette, key: Buchstabe): Zeichenkette`*.*

**Aufgabe 8.2** *Implementiere eine Operation* `skytale(data: Zeichenkette, key: Ganzzahl): Zeichenkette`*.*

**Aufgabe 8.3** *In einer Klassenarbeit in der Sekundarstufe I gibt es die Notenstufen von 1 bis 6. Schreiben Sie einen Block, der zu einer gegebenen Verteilung von Noten (als Liste mit sechs Elementen übergeben) ein Häufigkeitsdiagramm zeichnet.*

*Zusätzlich soll der Mittelwert berechnet und ausgegeben werden.*

**Aufgabe 8.4** *Ein Lehrer zensiert nach folgendem Verfahren: Er wirft drei Laplace-Würfel, addiert die Augenzahlen und substrahiert 3. Wie sind die einzelnen Punktergebnisse nach diesem Verfahren verteilt? Implementiere eine Operation, die ein entsprechendes Häufigkeitsdiagramm berechnet und zeichnet.*

**Aufgabe 8.5** *Erläutere den Unterschied zwischen einer Geheimschrift und einem Kryptosystem.*

*Wann ist ein Kryptosystem sicher?*

*In welche Gruppen von Kryptosystemen lassen sich Caesar und Skytale einordnen?*

**Aufgabe 8.6** *Implementiere ein Programm zum Knacken von Kryptotexten. Das Programm soll eine Häufigkeitsverteilung eines Kryptotextes zeichnen und dann in der Lage sein, eine Cäsar- bzw. eine Skytaleverschüsselung zu knacken. Verwende dazu die Blöcke aus den Aufgaben 9.1 und 9.2.*

# 9. Häufige Fehler

## 9.1. Wertzuweisungen

Der häufigste Fehler ist die Verwendung falscher Befehle bei der Wertzuweisung. Die folgende Tabelle zeigt die Zuordnung der Befehle zu den einzelnen Variablentypen. Dabei muss unterschieden werden, ob eine Variable für sich steht oder Element einer Liste ist:

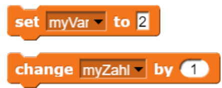



Abbildung 9.1.: Set und Change

Für einfache Variable, egal ob vom Typ Wahrheitswert, Zahl oder Text (String) verwenden wir den `SET`-Befehl, wobei für Zahlvariable auch der `CHANGE`-Befehl verwendet werden kann:

Häufig wird auch die Richtung im `SET`-Befehl vertauscht. Grundsätzlich folgt auf das `SET` zunächst die Variable, der ein Wert zugewiesen wird. Nach dem `TO` folgt dann der Wert, je nach Verwendungszweck als Wahrheitswert, Zahl, Zeichenkette oder Variable.

Liegt dagegen ein Element einer Liste vor, so verwenden wir den `REPLACE`-Befehl:

Abbildung 9.2.: Replace

Wird ein `SET`-Befehl für eine Liste durchgeführt, z.B. `set Liste to Alpha`, dann wird die Liste gelöscht und die Variable namens Liste mit dem unter to angegebenen Inhalt gefüllt, d.h. in diesem Fall mit `Alpha`. Aus einer Liste wird so eine Zeichenkette.

Die Verwendung eines falschen Blocks kann auch für einfache Variable schwerwiegende Folgen haben, weil Snap! keine durchgehende Prüfung der Typen durchführt.

Beispiel: MyVar habe den Wert „Zahl". Nun soll eine Eins hinten angefügt werden. Bei einem `change myVar by 1` geht Snap! davon aus, dass es sich um eine Zahlvariable handelt. Ein etwa vorhandener Text wird gelöscht, die Variable auf 0 gesetzt und dann 1 addiert. MyVar hat also anschließend den Wert 1.

## 9.2. Händische Inkrementierung der FOR-Schleife

Eine `FOR`-Schleife initialisiert vor Beginn die Zählvariable und inkrementiert nach jedem Durchgang die Zählvariable. Wird die Zählvariable vor der Schleife von Hand initialisiert, so ist diese Anweisung überflüssig, führt aber nicht zu einem Fehler. Wird jedoch die Zählvariable innerhalb der `FOR`-Schleife erhöht, dann werden nicht mehr alle Werte der Zählvariable angenommen.

```
for i=1 to 10
    add i to result
    change i by 1
```

liefert nicht - wie geplant - die Zahlen von 1 bis 10 - in result, sondern lediglich die Zahlen 1, 3, 5, 7, 9.

Man kann diesen Effekt bewusst ausnutzen, um z.B. nur jeden 2. Wert der Zählvariable zu betrachten. In allen anderen Fällen führt eine händische Inkrementierung zu Fehlern.

## 9.3. FOR-Schleife läuft rückwärts

Wir setzen die `FOR`-Schleife normalerweise so ein, dass der Startwert kleiner ist als der Endwert. In diesem Fall wird die Zählvariable hochgezählt. Der FOR-Block aus Snap! ist so programmiert, dass er prüft, ob der Startwert kleiner ist als der Endwert und dann hochzählt. Ist allerdings der Startwert größer als der Endwert,

dann zählt die Schleife herunter. Bei `FOR i=6 TO 2` nimmt die Zählvariable nacheinander die Werte 6, 5, 4, 3, 2 an. Diese Eigenschaft kann zu unerwünschten Nebenwirkungen führen, wenn wir als Endwert eine Differenz einfügen.

Nehmen wir als Beispiel eine Operation `loescheAb` mit den Parametern `wort` und `stelle`, die in `wort` alle Buchstaben ab `stelle` löscht. Wir setzen das neue Wort buchstabenweise zusammen bis `stelle-1`.

Abbildung 9.3.: loescheAb(wort, stelle) fehlerhaft

Für Werte von `stelle` ab 2 funktioniert die Operation wie erwartet. Allerdings sollte die Operation für `stelle=1` das ganze Wort löschen. Tatsächlich bleibt der erste Buchstabe erhalten. `loescheAb(ABCD,1` liefert `A`.

Abbildung 9.4.: Result von loescheAb

Das liegt daran, dass die `FOR`-Schleife hier einen Endwert erhält, der kleiner ist als der Startwert, nämlich `FOR i=1 TO 0`. In diesem Fall zählt die `FOR`-Schleife rückwärts, also 1, 0. Der erste Buchstabe wird an `result` angehängt, einen nullten Buchstaben gibt es nicht. Deshalb wird `A` ausgegeben.

Man kann diesen Fehler vermeiden, indem man die FOR-Schleife in eine einseitige Verzweigung einsetzt, die nur ausgeführt wird, wenn `stelle` größer ist als Eins.

Abbildung 9.5.: loescheAb korrigiert

## 9.4. Doppelkreuz in der Kopfzeile von Hand

Abbildung 9.6.: Ziehe

Bei manchen Blöcken findet sich in der Kopfzeile hinter Parametern ein Doppelkreuz. Dies bedeutet, dass für den Parameter nur Zahlen eingesetzt werden dürfen. Dieses Doppelkreuz darf - ebenso wie andere Variablentypen, z.B. drei Punkte für Listen - **nie von Hand eingegeben** werden. Vielmehr klickt man den Parameter an, wählt den kleinen schwarzen Pfeil links und stellt im **Edit Input Name** Dialog den gewünschten Typ ein, z.B. Number für Zahlen. Es gibt zwei Möglichkeiten, zu überprüfen, ob der Typ richtig eingestellt wurde. Wenn man den betreffenden Parameter nach unten zieht, verschwindet das Doppelkreuz. So lautet der Parameter im ersten Move-Block nicht `rechts #`, sondern schlicht `rechts`,

Eine zweite Kontrollmöglichkeit bietet der Block selbst. Eingabefelder für Zahlen sind durch ein ovales Feld gekennzeichnet. Ist das Feld eckig, wurde der Typ nicht richtig eingestellt.

Abbildung 9.7.: Ziehe

Abbildung 9.8.: Ziehe-Block

## 9.5. Verstoß gegen die Datenkapselung

Snap!-Programmierer sind gewohnt, alle Listenoperationen zur Verfügung zu haben. In anderen Programmiersprachen ist der Zugriff eingeschränkter. Z.B. kann man auf ein Listenelement erst dann zugreifen, wenn man, beginnend mit dem ersten Element der Liste, sich „durchgehangelt“ hat. Auch wenn wir ADTs mit Snap!-Listen implementieren, gilt für alle ADTs das Prinzip der Datenkapselung. Auf die Inhalt des ADT kann nur mit den Methoden des ADT zugegriffen werden, nicht mit den Snap!-Listenbefehlen!

## 9.6. Addieren mit dem ADD-Block

Der ADD-Block dient **ausschließlich** dazu, **ein Element hinten an eine Liste anzuhängen**.

Er kann **nicht** dazu benutzt werden, um zwei Zahlen zu addieren oder zwei Zeichenketten zu verbinden! Leider legt der Name des Blocks eine solche Fehlinterpretation nahe.

## 9.7. Parameter

Parameter werden beim Aufruf eines Blocks übergeben. Sie stehen im Quelltext in Klammern in der Kopfzeile. Dabei stehen in der Kopfzeile die Typen hinter den Parametern (Beispiel: `encode (data: Liste. key: Zeichenkette): Zeichenkette`). Die Typenbezeichnungen stehen **ausschließlich** in der Kopfzeile, nie im weiteren Verlauf des Quelltextes. Parameter stehen im weiteren Verlauf innerhalb des Blockes zur Verfügung. Sie werden deshalb **nicht bei den Skriptvariablen aufgezählt**.

Es ist weder notwendig noch sinnvoll, den Wert von Parametern innerhalb ihres Blocks durch eine Benutzerabfrage oder eine Zufallsfunktion festzulegen.

## 9.8. Alleinstehender Reporter

Methoden zur Anzeige der Eigenschaften von Objekten haben häufig Namen, die aus einem Verb bestehen wie `join`, `split ... by` oder `pop`, oder die mit einem Verb zusammengesetzt sind, wie `inhaltGeben` oder `getLength`. Dennoch handelt es sich um Reporter, die in Snap! als Oval dargestellt werden. Die Form deutet bereits darauf hin, dass ein solcher Block nicht allein in einer Zeile stehen darf. Das dürfen nur Anweisungen (Command), die als Puzzleteile dargestellt werden.

Beispiel: `Schlange.dequeue()` kann nicht allein in einer Zeile stehen, wohl aber als Parameter in einer Wertzuweisung z.B.

```
S1.enqueue(S2.dequeue)
```

Gleiches gilt auch für Blöcke des Typs Prädikat, z.B. `is Liste empty?`. Auch sie können nicht allein in einer Zeile stehen.

## 9.9. Verwendung der deutschen Sprache

Das Kerncurriculum Informatik schreibt vor, dass für bestimmte Datentypen die deutsche Bezeichnung verwendet wird, z.B. Reihung für ARRAY, Zeichenkette für STRING, Wahrheitswert für BOOLEAN. **Als Programmierspra-**

**che verwenden wir grundsätzlich Englisch**.

## 9.10. Struktogramm-Schreibweisen im Quelltext

Bestimmte Schreibweisen wie der Zugriff auf Listenelement über den Index in eckigen Klammern oder der Zuweisungspfeil sind **ausschließlich in Struktogrammen zulässig**. Wenn die Arbeitsanweisung lautet „Implementieren Sie eine Operation, die ...“, dann ist Quelltext verlangt. Statt Liste[$i$] ist dann zu schreiben `item i of Liste` und an die Stelle des Zuweisungspfeils tritt `set Variable to Wert`. Bitte beachten Sie dabei, dass die Variable der erste Parameter ist und Wert der zweite.

## 9.11. Kopieren von Listen

Eine gefährliche Fehlerquelle ist es, wenn Listen durch einfache Zuweisung kopiert werden. In diesem Fall erstellt Snap! keine echte Kopie. Vielmehr erkennt es, dass es sich um eine Liste handelt und gibt lediglich die Startadresse weiter. In der Folge wird also eine Liste durch verschiedene Variable angesprochen. Änderungen in der einen Liste tauchen automatisch in der anderen auf.

Abbildung 9.9.: 2 verbundene Listen

Um solche Effekte zu vermeiden, muss man mit einer `FOR`-Schleife alle Elemente der Liste auf die neue übertragen. Dann entsteht eine echte Kopie, die sich durch Lösch- und Hinzufüge-Befehle auch vom Original unterscheiden kann.

Alternativ kann man auch den MAP-Befehl verwenden: `set 2ndList to map ringify ( ) over myList` erzeugt eine echte Kopie und weist sie der Variablen 2ndList zu.

# 10. Liste der Snap!-Blöcke

Das folgende Kapitel enthält die Snap!-Blöcke, die in diesem Skript verwendet werden, geordnet nach den Kapiteln ihres ersten Auftretens.

## 10.1. Grundlagen der Programmierung

| Blöcke | Bedeutung |
|---|---|
| when [Flagge] clicked | grüne Flagge (Start) |
| when space key pressed | Wenn Space-Taste gedrückt wird |
| when I receive | Wenn ich eine Nachricht erhalte |
| if | einseitige Verzweigung |
| if … else | zweiseitige Verzweigung |
| repeat 10 | wiederholt die eingeschlossenen Anweisungen |
| repeat until | Wiederholung mit vorprüfender Abbruchbedingung |
| for i = 1 to 10 | FOR-Schleife initialisiert die Zählvariable, führt die Wiederholungen aus und inkrementiert die Zählvariable nach jedem Durchgang |

| Blöcke | Bedeutung |
|---|---|
| script variables a | ermöglicht die Erzeugung von lokalen Variablen |
| stop all | beendet alle Skripte |

## 10.2. Werkzeuge

| Blöcke | Bedeutung |
|---|---|
| forever | Endlosschleife |
| touching mouse-pointer ? | berührt Mauszeiger / Rand / Zeichenspur |

## 10.3. Grafik

| Blöcke | Bedeutung |
|---|---|
| move 10 steps | bewegt das aktuelle Sprite die angegebene Anzahl von Pixeln in die aktuelle Richtung. |
| turn ↻ 15 degrees | dreht das aktuelle Sprite um die Gradzahl nach rechts. |
| point in direction 90 | dreht das aktuelle Sprite in die angegebene Richtung, falls es noch nicht dahin zeigt. |
| point towards mouse-pointer | dreht das aktuelle Sprite in Richtung auf eine zufällige Position / den Mauszeiger / den Bildschirmmittelpunkt |

| Blöcke | Bedeutung |
|---|---|
| go to x: 0 y: 0 | bewegt das aktuelle Sprite an die angegebene Position. |
| go to random position | bewegt das aktuelle Sprite auf eine zufällige Position / den Mauszeiger / den Mittelpunkt des Bildschirms |
| warp | Warp unterbricht alle anderen Skripte und das automatische Update der Stage. Nach Ablauf werden alle Änderungen auf ein Mal vorgenommen. |
| run | Run führt die Blöcke innerhalb der Umrandung (ringify) aus |
| | ringify ermöglicht die Übergabe des eingeschlossenen Blocks |
| switch to costume | wechselt zu einem neuen Kostüm |
| new costume width height | erzeugt aus einer Farbliste ein neues Kostüm in den angegebenen Maßen |
| RGBA at mouse-pointer | zeigt den RGBA-Wert eines Punktes |
| clear | löscht die Stage |
| fill | füllt die Zeichenfläche bis zum nächsten Rand mit der aktuellen Farbe |
| pen down | setzt den Zeichenstift ab |
| pen up | hebt den Zeichenstift an |
| pick random 1 to 10 | bestimmt eine Zufallszahl in den angegebenen Grenzen |
| list | liefert eine Liste. In die freien Felder kann der Inhalt eingesetzt werden |
| add thing to | fügt den Inhalt als letztes Element an die angegebene Liste an |

## 10.4. Mathematische Algorithmen

| Blöcke | Bedeutung |
|---|---|
| + <br> − <br> × <br> / | Rechenoperationen Plus, Minus, Mal und Geteilt durch |
| mod | liefert den ganzzahligen Rest bei Division der ersten durch die zweite Zahl |
| round | rundet eine Zahl ab oder auf |
| sqrt of 10 | zieht die Wurzel |
| neg of 10 | liefert die Gegenzahl |
| abs of 10 | liefert den Betrag einer Zahl |
| ceiling of 10 | rundet eine Zahl auf bis zur nächsten ganzen Zahl |
| floor of 10 | rundet eine Zahl ab bis zur nächsten ganzen Zahl |
| < <br> = <br> > | Vergleichsoperationen Größer, Gleich und Kleiner |
| true false | liefert einen Wahrheitswert |
| is 5 a number ? | prüft, ob der Parameter eine Zahl ist |
| set to 0 | weist einer Variable einen Wert oder eine Liste zu |
| change by 1 | ändert den Wert einer Zahlvariablen |

## 10.5. Dualzahlen

| Blöcke | Bedeutung |
|---|---|
| ( ) ^ ( ) | liefert eine Potenz |

## 10.6. Umgang mit Zeichenketten

| Blöcke | Bedeutung |
|---|---|
| ask what's your name? and wait | ermöglicht die Eingabe einer Zeichenkette |
| unicode 65 as letter | liefert den Unicode eines Zeichen |
| unicode of a | liefert das Zeichen zu einem Unicode |
| length of text world | liefert die Länge einer Zeichenkette |
| letter 1 of world | gibt das angegebene Zeichen einer Zeichenkette |
| [ ] = [ ] | unterscheidet nicht zwischen Groß- und Kleinschreibung, solange links und rechts jeweils Zeichenketten stehen |
| join hello world | fügt die Parameter zu einer Zeichenkette zusammen |

## 10.7. Sortieren

| Blöcke | Bedeutung |
|---|---|
| item 1 of | liefert das angegebene Element einer Liste |
| length of | gibt die Anzahl der Elemente in einer Liste an |
| length of | gibt die Anzahl der Elemente in einer Liste an |

| Blöcke | Bedeutung |
|---|---|
| delete 1 of | löscht ein Element aus einer Liste an |
| insert thing at 1 of | fügt ein Element in eine Liste ein |
| replace item 1 of with thing | ersetzt ein Element in einer Liste |
| contains thing | prüft, ob ein Element in einer Liste enthalten ist |
| is empty? | prüft, ob eine Liste leer ist |

## 10.8. Verschlüsselungsverfahren

| Blöcke | Bedeutung |
|---|---|
| numbers from 1 to 10 | liefert eine Zahlenliste von Startwert bis Endwert |
| write Hello! size 12 | schreibt den angegebenen Text in der gewünschten Größe auf den Bildschirm |

## 10.9. Blöcke aus NetsBlox

Bei Wahl von `Message` öffnet sich ein weiteres Fenster für die Botschaft.

| Blöcke | Bedeutung |
|---|---|
| send msg to | sendet eine Botschaft unter NetsBlox |
| when I receive | empfängt eine Botschaft unter NetsBlox |

# 11. Übungsklausur

Gegeben ist das folgende Snap!-Programm mit drei Blöcken `rechteck`, `hoch` und `figur`. Der Operator * steht für die Multiplikation:

```
rechteck(breite, hoehe)
pen down
repeat 2
   move breite steps
   turn left 90 degrees
   move hoehe steps
   turn left 90 degrees
pen up
```

```
  hoch(a)
turn left 90 degrees
move a steps
turn right 90 degrees
```

```
  figur(a)
rechteck(5*a, 7*a)
move a steps
rechteck(a, 2*a)
move 2*a steps
hoch(a)
rechteck(a, a)
repeat 2
   move -2*a steps
   hoch(2*a)
   rechteck(a, a)
   move 2*a steps
   rechteck(a, a)
hoch(-5*a)
move 2*a steps
```

Hauptprogramm:

```
When Start clicked
clear
show
goto x: -100 y:-100
point in direction 90
figur(20)
```

**Aufgabe 11.1** *a) Erläutere die ersten vier Zeilen des Hauptprogramms (beginnend mit **`clear`**).*

*b) Erstelle eine Zeichnung, die wiedergibt, was das Programm auf den Bildschirm zeichnet (Tipp: 1 Kästchen 10 Einheiten). Notiere, wo der Cursor am Ende des Programms steht.*

*c) Auf das Gebäude soll ein Dach gesetzt werden. Erläutere, welche Veränderungen man im Programm vornehmen muss.*

Gegeben ist die folgende Operation:

```
pos(wort: Zeichenkette, b: Buchstabe):
Ganzzahl
script variables i, result
set result to 0
for i=1 to length of text wort
   if letter i of wort = b
      report i
report result
```

**Aufgabe 11.2** *a) Erstelle Tracetabellen für `pos(KLAUSUR, U)` und `pos(HAUS, E)`.*

*b) Erläutere, wie die Operation pos funktioniert und was sie ausgibt.*

*c) Implementiere eine Operation `last(wort, buchstabe)`, die angibt, an welcher Position ein Buchstabe das letzte Mal in dem Wort vorkommt. Beispiel: `last(ESEL, E)` liefert die Zahl 3.*

zu 11.1 a:

`clear` löscht die Zeichenspuren vom Bildschirm

`show` zeigt das aktuelle Sprite

`goto x: -100 y: 100` geht an die angegebene Position

`point in direction 90` wenn das Sprite nicht in die angegebene Richtung zeigt, dreht es sich in diese Richtung.

zu 11.1 b:

Es zeichnet ein „Hochhaus“ mit drei Stockwerken. Der Cursor steht am Ende an

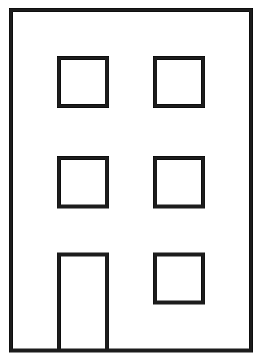


Abbildung 11.1.: zu Aufgabe 14.6c

der rechten unteren Ecke und zeigt nach rechts.

zu 11.1 c:
am Schluss muss ein Dreieck eingefügt werden, z.B.

```
hoch (7*a)
turn left 135
pen down
move sqrt(12.5)*a
turn left 90
move sqrt(12.5)*a
turn left 135
pen up
move 5*a
hoch (-7*a)
```

zu 11.2 a:
wort = KLAUSUR; b = U

| result | i | letter i of wort |
|---|---|---|
| 0 | | |
| | 1 | |
| | | K |
| | 2 | |
| | | L |
| | 3 | |
| | | A |
| | 4 | |
| | | U |

**report 4**

wort = HAUS, b = E

| result | i | letter i of wort |
|---|---|---|
| 0 | | |
| | 1 | |
| | | H |
| | 2 | |
| | | A |
| | 3 | |
| | | U |
| | 4 | |
| | | S |

**report 0**

zu 11.2 b:
Die Operation `pos` geht mit einer `FOR`-Schleife zeichenweise das als Parameter übergebene Wort durch und prüft, ob der aktuelle Buchstabe gleich dem als zweiten Parameter übergebene Buchstabe ist. Falls Gleichheit vorliegt, wird die aktuelle Stelle ausgegeben und der Block beendet. Wurde die `FOR`-Schleife komplett abgearbeitet, ohne dass der Buchstabe gefunden wurde, wird statt dessen 0 ausgeben. Die Operation `pos` gibt also aus, an welcher Stelle ein Buchstabe zum ersten Mal in dem Wort vorkommt.

zu 11.2 c:
Die Operation `pos` ist so aufgebaut, dass beim ersten Auffinden des Buchstabens ein Report erfolgt, d.h. dass der Block damit beendet wird. Wenn wir statt dessen jedes Mal bei Gleichheit den aktuellen Wert der Zählvariable in result speichern und erst nach dem Durchlaufen ganzen der FOR-Schleife ausgeben, haben wir das letzte Vorkommen des Buchstabens gespeichert.

```
  last(wort: Zeichenkette, b:
Buchstabe): Ganzzahl
script variables i, result
set result to 0
for i=1 to length of text wort
   if letter i of wort = b
     set result to i
report result
```

# A. ASCII-, Hexadezimal- und Dualzahlen-Tabelle

Tabelle A.1.: Dezimal-, Hexadezimal- und Dualzahlen, ASCII-Code

| dez | hex | dual | ASCII |
|---|---|---|---|
| 0 | 00 | 000 0000 | NUL |
| 1 | 01 | 000 0001 | SOH |
| 2 | 02 | 000 0010 | STX |
| 3 | 03 | 000 0011 | ETX |
| 4 | 04 | 000 0100 | EOT |
| 5 | 05 | 000 0101 | ENQ |
| 6 | 06 | 000 0110 | ACK |
| 7 | 07 | 000 0111 | BEL |
| 8 | 08 | 000 1000 | BS |
| 9 | 09 | 000 1001 | TAB |
| 10 | 0A | 000 1010 | LF |
| 11 | 0B | 000 1011 | VT |
| 12 | 0C | 000 1100 | FF |
| 13 | 0D | 000 1101 | CR |
| 14 | 0E | 000 1110 | SO |
| 15 | 0F | 000 1111 | SI |
| 16 | 10 | 001 0000 | DLE |
| 17 | 11 | 001 0001 | DC1 |
| 18 | 12 | 001 0010 | DC2 |
| 19 | 13 | 001 0011 | DC3 |
| 20 | 14 | 001 0100 | DC4 |
| 21 | 15 | 001 0101 | NAK |
| 22 | 16 | 001 0110 | SYN |
| 23 | 17 | 001 0111 | ETB |
| 24 | 18 | 001 1000 | CAN |
| 25 | 19 | 001 1001 | EM |
| 26 | 1A | 001 1010 | SUB |
| 27 | 1B | 001 1011 | Esc |
| 28 | 1C | 001 1100 | FS |
| 29 | 1D | 001 1101 | GS |
| 30 | 1E | 001 1110 | RS |
| 31 | 1F | 001 1111 | US |
| 32 | 20 | 010 0000 | SP |
| 33 | 21 | 010 0001 | ! |
| 34 | 22 | 010 0010 | “ |
| 35 | 23 | 010 0011 | # |
| 36 | 24 | 010 0100 | $ |
| 37 | 25 | 010 0101 | % |
| 38 | 26 | 010 0110 | & |
| 39 | 27 | 010 0111 | ’ |
| 40 | 28 | 010 1000 | ( |
| 41 | 29 | 010 1001 | ) |
| 42 | 2A | 010 1010 | * |
| 43 | 2B | 010 1011 | + |
| 44 | 2C | 010 1100 | , |
| 45 | 2D | 010 1101 | - |
| 46 | 2E | 010 1110 | . |
| 47 | 2F | 010 1111 | / |
| 48 | 30 | 011 0000 | 0 |
| 49 | 31 | 011 0001 | 1 |
| 50 | 32 | 011 0010 | 2 |
| 51 | 33 | 011 0011 | 3 |
| 52 | 34 | 011 0100 | 4 |
| 53 | 35 | 011 0101 | 5 |
| 54 | 36 | 011 0110 | 6 |
| 55 | 37 | 011 0111 | 7 |
| 56 | 38 | 011 1000 | 8 |
| 57 | 39 | 011 1001 | 9 |
| 58 | 3A | 011 1010 | : |
| 59 | 3B | 011 1011 | ; |
| 60 | 3C | 011 1100 | ¡ |
| 61 | 3D | 011 1101 | = |
| 62 | 3E | 011 1110 | ¿ |
| 63 | 3F | 011 1111 | ? |
| 64 | 40 | 100 0000 | @ |
| 65 | 41 | 100 0001 | A |
| 66 | 42 | 100 0010 | B |
| 67 | 43 | 100 0011 | C |
| 68 | 44 | 100 0100 | D |
| 69 | 45 | 100 0101 | E |
| 70 | 46 | 100 0110 | F |
| 71 | 47 | 100 0111 | G |
| 72 | 48 | 100 1000 | H |
| 73 | 49 | 100 1001 | I |
| 74 | 4A | 100 1010 | J |
| 75 | 4B | 100 1011 | K |
| 76 | 4C | 100 1100 | L |

| dez | hex | dual | ASCII |
|---|---|---|---|
| 77 | 4D | 100 1101 | M |
| 78 | 4E | 100 1110 | N |
| 79 | 4F | 100 1111 | O |
| 80 | 50 | 101 0000 | P |
| 81 | 51 | 101 0001 | Q |
| 82 | 52 | 101 0010 | R |
| 83 | 53 | 101 0011 | S |
| 84 | 54 | 101 0100 | T |
| 85 | 55 | 101 0101 | U |
| 86 | 56 | 101 0110 | V |
| 87 | 57 | 101 0111 | W |
| 88 | 58 | 101 1000 | X |
| 89 | 59 | 101 1001 | Y |
| 90 | 5A | 101 1010 | Z |
| 91 | 5B | 101 1011 | [ |
| 92 | 5C | 101 1100 | \ |
| 93 | 5D | 101 1101 | ] |
| 94 | 5E | 101 1110 | ˆ |
| 95 | 5F | 101 1111 | _ |
| 96 | 60 | 110 0000 | ’ |
| 97 | 61 | 110 0001 | a |
| 98 | 62 | 110 0010 | b |
| 99 | 63 | 110 0011 | c |
| 100 | 64 | 110 0100 | d |
| 101 | 65 | 110 0101 | e |
| 102 | 66 | 110 0110 | f |
| 103 | 67 | 110 0111 | g |
| 104 | 68 | 110 1000 | h |
| 105 | 69 | 110 1001 | i |
| 106 | 6A | 110 1010 | j |
| 107 | 6B | 110 1011 | k |
| 108 | 6C | 110 1100 | l |
| 109 | 6D | 110 1101 | m |
| 110 | 6E | 110 1110 | n |
| 111 | 6F | 110 1111 | o |
| 112 | 70 | 111 0000 | p |
| 113 | 71 | 111 0001 | q |
| 114 | 72 | 111 0010 | r |
| 115 | 73 | 111 0011 | s |
| 116 | 74 | 111 0100 | t |
| 117 | 75 | 111 0101 | u |
| 118 | 76 | 111 0110 | v |
| 119 | 77 | 111 0111 | w |
| 120 | 78 | 111 1000 | x |
| 121 | 79 | 111 1001 | y |
| 122 | 7A | 111 1010 | z |
| 123 | 7B | 111 1011 | { |
| 124 | 7C | 111 1100 | — |
| 125 | 7D | 111 1101 | } |
| 126 | 7E | 111 1110 | ˜ |
| 127 | 7F | 111 1111 | DEL |

Tabelle A.2.: Unicode für deutsche Umlaute

| dez | hex | dual | Unicode |
|---|---|---|---|
| 196 | C4 | 1100 0100 | Ä |
| 214 | D6 | 1101 0110 | Ö |
| 220 | DC | 1101 1100 | Ü |
| 223 | DF | 1101 1111 | ß |
| 228 | E4 | 1110 0100 | ä |
| 246 | F6 | 1111 0110 | ö |
| 252 | FC | 1111 1100 | ü |