BYOB-Kompendium

Fritz Hasselhorn

16. September 2015

Inhaltsverzeichnis

[1] Jona Eppmann
[2] Julian Comte
[3] Jasper Freese
[4] Mirco Troue

[5] Daniel Marx
[6] Niklas Euler

Kapitel 1

Elementare Kontrollstrukturen

In der Informatik unterscheiden wir unabhängig von der Programmiersprache verschiedene elementare Kontrollstrukturen wie Anweisung, Sequenz, Verzweigung und Schleife.

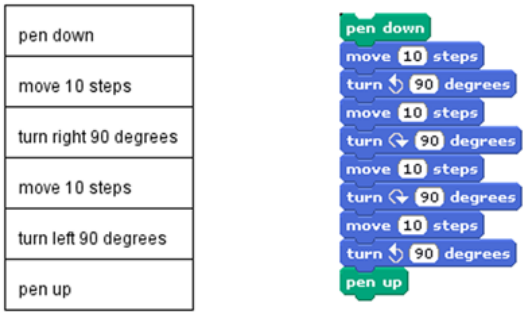
1.1 Anweisung

Eine einzelne Anweisung bedeutet, dass der Computer einen Befehl ausführt, z.B. eine Bewegung oder eine sonstige Aktion. In BYOB haben Blöcke, die als Anweisung ausgeführt werden können, die Form eines Puzzleteils, um anzudeuten, dass diese Anweisungen zusammengesetzt werden können.

Im Struktogramm wird die Anweisung als Rechteck gezeichnet:

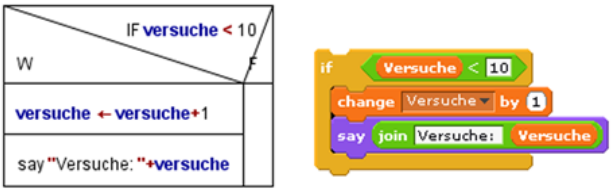
1.2 Sequenz

Eine Sequenz besteht aus mehreren Anweisungen, die nacheinander ausgeführt werden:

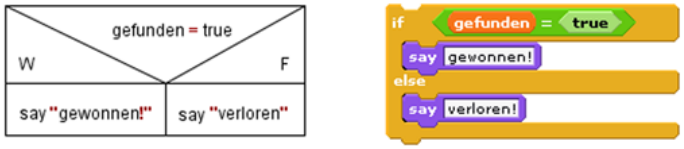


1.3 Verzweigung

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 einseitige Verzweigung 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.

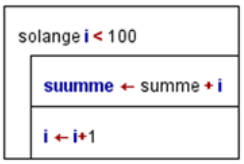


1.4 Schleife

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

1.4.1 Vorprüfende Schleife

Eine vorprüfende Schleife, auch WHILE-Schleife genannt, prüft zunächst eine Bedingung, bevor er Code ausführt, daher auch der Name. Wird die Bedingung erfüllt, so wird der Codeabschnitt innerhalb der Schleife ausgeführt. Anschließend wird wiederum die Bedingung erfüllt und falls zutreffend der Codeinhalt innerhalb der Schleife ausgeführt, usw. An dieser Stelle muss die Programmierin 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 Wort „WHILE (Wiederholungsbedingung) DO ... “. Daher kommt die Bezeichnung WHILE-Schleife. BYOB verfügt über keinen WHILE-Block. Der „REPEAT UNTIL (Bedingung) ... “-Block 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 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 (vgl. Gesetze der Schaltalgebra):

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

und

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

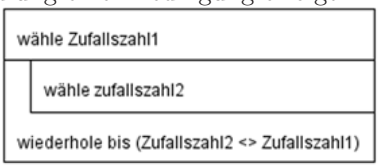
Das bedeutet, dass die Verneinung einer AND-Verknüpfung auch dargestellt werden kann, indem die beiden Einzelbedingungen verneint und dann mit OR verknüpft werden. Die Verneinung einer OR-Verknüpfung kann auch dargestellt werden, indem die beiden Einzelbedingungen verneint und dann mit AND verknüpft werden. Beispiele:

$$\overline{(vertauscht = true) \wedge (versuche < 11))} = \overline{vertauscht = true} \vee \overline{versuche < 11}$$

$$\overline{(vertauscht = true) \vee (versuche < 11)} = \overline{vertauscht = true} \wedge \overline{versuche < 11}$$

1.4.2 Nachprüfende Schleife

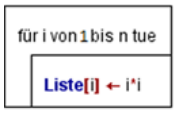
Eine nachprüfende Schleife wird in jedem Fall einmal ausgeführt, bevor die Prüfung einer Bedingung erfolgt.



Beispiel: Zunächst wird eine Zufallszahl1 gewählt, dann wird eine Zufallszahl2 gewählt und dies wird so lange wiederholt, bis die Abbruchbedingung erfüllt ist, d.h. bis beide Zufallszahlen verschieden sind. In Delphi heißt diese Schleife die REPEAT-UNTIL-Schleife, während die vorprüfende Schleife als WHILE-DO-Schleife bezeichnet wird. In BYOB ist keine nachprüfende Schleife enthalten. Zur Realisierung muss zunächst der Quelltext einmal ausgeführt werden, dann erfolgt die Prüfung der Abbruchbedingung und ggf. weitere Wiederholungen:

1.4.3 Zählschleife

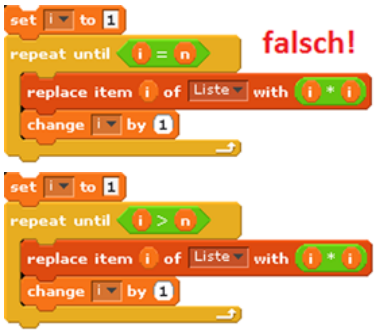
Eine Zählschleife oder auch FOR-Schleife leistet drei Dinge:
Zunächst wird die Zählvariable auf den Startwert gesetzt. Dann wird der Codeabschnitt im Inneren der Schleife ausgeführt. Anschließend wird die Zählvariable um Eins erhöht. Es schließt sich die nächste Wiederholung an mit Ausführung und usw. Wenn die Zählvariable den Endwert erreicht hat, wird der Code noch einmal ausgeführt. Dann wird die Schleife verlassen.



Beachtenswert ist die Anzahl der Wiederholungen. Nehmen wir an, eine Zählschleife zählt von 2 bis 5. Dann ist 2 der Startwert, 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 BYOB, so muss man drei Dinge tun:

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

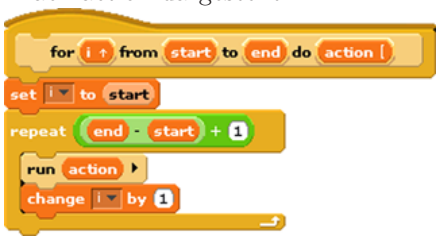
Achtung! Verwendet man statt der REPEAT-Schleife unter BYOB eine REPEAT-UNTIL-Schleife, dann muss man die obere Grenze so wählen, dass der Codeabschnitt auch mit dem Endwert noch durchgeführt wird. Statt



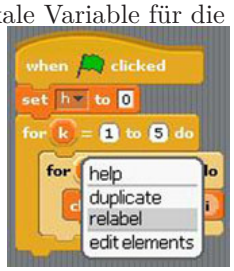
wäre richtig

1.4.4 FOR-Schleife selbst gebaut

BYOB verfügt standardmäßig über keine FOR-Schleife, wie sie in anderen Programmiersprachen wie z.B. Delphi oder Java vorhanden ist. Eine solche Schleife kann aber einfach selbst erstellt und dann eingesetzt werden. Dazu erstellen wir eine neuen Command-Block der Kategorie „Control“. Wir nennen den Block `"For %i from %start to %end do %action"`. Er verfügt also über die vier Parameter i, start, end und action. Für i wählen wir im Parametertyp-Menü die Option „make internal variable visible to caller“. Im Kopf der Blocks wird das durch einen Pfeil nach oben hinter dem i dargestellt. Für start und end wollen wir nur Zahlen als Eingabe zulassen, wir wählen also zwei Mal die Option „number“. Unserer FOR-Schleife soll die Möglichkeit bieten, andere Anweisungen einzufügen, wie z.B. bei REPEAT- oder REPEAT-UNTIL-Schleifen. Deshalb wählen wir für action den Typ C-Shape und packen die Variable in einen RUN-Block. Im Blockkopf wird dieser durch eine öffnende eckige Klammer nach action dargestellt:



Im Kontextmenü können wir unter „RELABEL“ die gewünschte globale oder lokale Variable für die Zählvariable einstellen:



ACHTUNG: Unbedingt ein relabel durchführen, wenn Sie die FOR-Schleife in einem eigenen Progamm einsetzen! Sonst produzieren Sie möglicherweise einen Programmabsturz!

Die FOR-Schleife ist in BYOB oft etwas hackelig. Deshalb hat sich folgende Vorgehensweise bewährt: Die FOR-Schleife anfügen, mit RELABEL eine Zähl-

variable auswählen und Start- und Endwert einstellen. Die Blöcke, die in die FOR-Schleife gehören, angehängt unter dem Block zusammenstellen und dann den fertigen Inhalt in die FOR-Schleife schieben.

Entsprechend können wir bei Bedarf auch eine FOR-Schleife DOWNTO erstellen, die herunter zählt.

Kapitel 2

Blöcke in BYOB

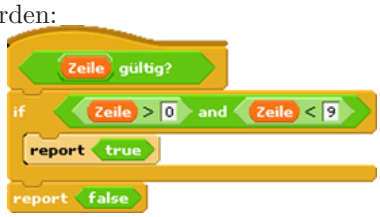
BYOB kennt drei Arten von Blöcken:

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 lassen sich keine weiteren Blöcke andocken.

Ein Report-Befehl liefert den angegebenen Rückgabewert und beendet dann den angegebenen 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.

2.2 Reporter

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

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

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:

Einfacher geht es, wenn einfach die Bedingung reportet wird:

Auch ein Prädikat kann nie allein in einer Programmzeile stehen.

2.4 Wertzuweisung in Struktogrammen

In Struktogrammen wird die Wertzuweisung häufig durch einen kleinen Linkspfeil dargestellt:

Wertzuweisung
zahl ← zahl * 2
name ← "Hildermeier"

Links steht die Variable, rechts der neuer Wert.

Kapitel 3

Variablen in BYOB

BYOB kennt verschiedene Arten von Variablen mit unterschiedlicher Reichweite.

3.1 Globale Variable

Über den Button „Make a variable“ 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).

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 also bei Parametern nicht notwendig, ihnen einen Startwert explizit zuzuweisen (Initialisieren von Variablen). Das geschieht beim Aufruf automatisch.

Parameter dürfen nicht in die Liste der Skriptvariablen aufgenommen werden. BYOB erzeugt in diesem Fall eine lokale Variable mit dem gleichen Namen. Im Kontextmenue 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.

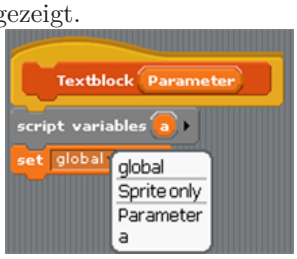
Man sollte vermeiden, Parametern den gleichen Namen zu geben wie globalen Variablen. Dann kann man nämlich innerhalb des Blocks nicht mehr auf die globalen Variablen zugreifen. Will man keine neuen Namen erfinden, so setzt man ein kleines p für Parameter vor den Namen.

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:

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.



Auch bei Skriptvariablen sollte man vermeiden, sie genau so zu benennen wie globale Variable. 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.

Verwendung der Variablenarten:

globale Variable	**Parameter**	**Skriptvariable**
sparsam verwenden	zur Anpassung von Blöcken	zur Kapselung von Daten
ggf. initialisieren set MyZahl to 1/0 set MyWort to „”	erhalten Startwert beim Aufruf	ggf. initialisieren set MyZahl to 1/0 set MyWort to „”
selbsterklärende Namen verwenden	selbsterklärende Namen verwenden	selbsterklärende Namen verwenden Ausnahme: Zählvariablen
	überschreiben globale Variable	überschreiben globale Variable

Kapitel 4

Datentypen

Die klassischen Programmiersprachen sind in einer Zeit entstanden, die geprägt war von der Knappheit an Speicherplatz. So entstanden z.B. unterschiedliche Datentypen für die Repräsentation von ganzen Zahlen (shortint, byte, integer), die sich vor allem durch ihren Speicherbedarfplatz unterscheiden.

4.1 Wahrheitswert (BOOLEAN)

Wahrheitswerte werden mit dem SET-Block eingestellt.
Sie können anschließend überall in alle sechseckigen Lücken eingesetzt werden. Dabei erfolgt keine Prüfung durch BYOB. Der Programmierer ist dafür verantwortlich, dass die Variable den Wert true oder false hat.

Achtung: Der Change-Block darf nicht auf Wahrheitswerte angewandt werden! Wird in dem oben angegebenen Beispiel `change(gefunden)by(1)` eingegeben, dann vermutet BYOB, dass es sich nicht um einen Wahrheitswert, sondern um eine Zahlvariable handelt, setzt den Inhalt auf Null und addiert Eins dazu.

4.2 Zahlen (INTEGER/REAL)

BYOB/SNAP! kann nur den Typ Zahl (Number). Es wird nicht zwischen Ganzzahlen und Kommazahlen unterschieden. Falls - wie ganz häufig in der Informatik - nur ganze Zahlen zugelassen sind, muss der Programmierer dies sicherstellen (dazu mehr in 5.1 Rund um den MOD-Block).

Eine Zahlvariable kann mit zwei verschiedenen Blöcken verändert werden:

Dabei kann der `change`-Block durch ein `set(MyVar)to(MyVar+1)` ersetzt werden.

Wird eine neue Variable vereinbar, so erhält sie automatisch den Wert Null.

Einige wichtige Blöcke für Zahlen:

`Abs (MyZahl)` liefert den Betrag einer Zahl.
`Round (MyZahl)` rundet eine Zahl ab.
`is(MyZahl) a number?` prüft, ob MyZahl eine Zahl (oder ein anderer Variablentyp ist.

Häufig benötigt man Dezimalstellen einer bestimmten Länge. Dazu multipliziert man mit der gewünschten Zählerpotenz, rundet und teilt dann wieder durch die Zehnerpotenz:

4.3 Buchstaben (CHAR)[1]

In Byob haben wir nur zwei Funktionen, die sich mit einzelnen Buchstaben, englisch `Chars` genannt, beschäftigen. Es handelt sich bei den beiden Funktionen um folgende:

Beide Funktionen arbeiten mit dem ASCII-Code, der eine 8-Bit Zeichencodierung ist. Jedem Textzeichen oder Buchstaben wird ein Bitmuster zugeordnet und da jedes Bitmuster auch als Dualzahl interpretiert werden kann, wandelt der Ascii-Code vereinfacht gesagt ein Zeichen in eine Zahl um. Beispiel: der ASCII-Wert von „A“ ist 65 und von „Z“ 90. Die Kleinbuchstaben beginnen bei „a“ gleich 97. Die Operation ASCII-Code liefert lediglich die Dezimalzahl des jeweiligem Zeichens. Man fügt dort einen Buchstaben ein und erhält einen Wert. Der Befehl `ASCII () as letter` gibt das Zeichen an, wenn man die Dezimalzahl dazu eingibt, d.h wenn man eine Zahl in das dazu zugehörige Satzzeichen übersetzen möchte, nutzt man diesen Befehl. Wenn man die Befehle ineinander setzt `(ASCII code of (ASCII(zahl) as letter))`, dann gibt Byob die Zahl aus.

In diesem Beispiel gibt Byob 48 als Lösung an.

Wenn man mehrere Buchstaben im `ASCII code of ()` eingibt, gibt es in Byob dafür keine Lösung, und es wird 0 ausgegeben, d.h die Befehle funktionieren nur mit einem einzelnen Buchstaben oder einer Zahl.

[1] Jona Eppmann

4.4 Zeichenketten (STRING)[2]

Von oben nach unten:

1. Ein String („Dies ist eine Zeichenkette!“) kann in einer Variable (string) gespeichert werden.
2. An den String können weitere Zeichen angehängt werden. („Dies ist eine Zeichenkette! Es funktioniert! “)
3. Es kann ein beliebiger Buchstabe, maximal die Anzahl der gesamten Buchstaben, aus diesem String ausgegeben werden (In diesem Fall ist der 4. Buchstabe das Leerzeichen).
4. Die Länge des Strings kann ausgegeben werden (27).

Für Zeichenketten dürfen nur die grünen Operatoren verwendet werden. Die braunen werden für Listen benötigt und sind somit unzulässig für Zeichenketten, es würden nur Fehler auftreten. Besonders gefährlich ist die Verwechslung des grünen und des braunen Blocks `length~of(...)`. In SNAP! werden Fehler angezeigt, in BYOB hingegen nicht. Zusätzlich gibt es in SNAP! einen Befehl, um einen String in einer Liste zu speichern, aufgeteilt durch Leerzeichen, Buchstaben usw..

Achtung: Der Change-Block darf nicht auf Zeichenketten angewandt werden! Wird in dem oben angegebenen Beispiel `change(string)by(1)` eingegeben, dann vermutet BYOB, dass es sich nicht um einen Zeichenkette, sondern um eine Zahlvariable handelt, setzt den Inhalt auf Null und addiert Eins dazu.

4.5 Listen (ARRAY)[3]

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

Listen können in BYOB erstellt werden, indem man eine globale Liste über den Button `Make a list` definiert oder indem man eine Variable über `Make a variable` oder eine Scriptvariable erstellt und dieser durch den Befehl `set MyList to list {}` eine Liste zuweist.

In Scriptvariablen erstellte Listen können jedoch nicht auf dem BYOB-Bildschirm betrachtet werden. Außerdem ist es nicht möglich, die in Variablen gespeicherten Listen auf der „Bühne“zu bearbeiten bzw alle Elemente durch

[2] Julian Comte
[3] Jasper Freese

Scrollen zu betrachten. Ansonsten sind beide Typen von Listen gleich zu betrachten. In SNAP! fällt die Möglichkeit, globale Listen zu definieren weg, weshalb hier die Zuweisung über globale oder Scriptvariablen genutzt werden muss. Zudem können hier die als Listen definierten globalen Variablen auf der „Bühne“ bearbeitet werden.

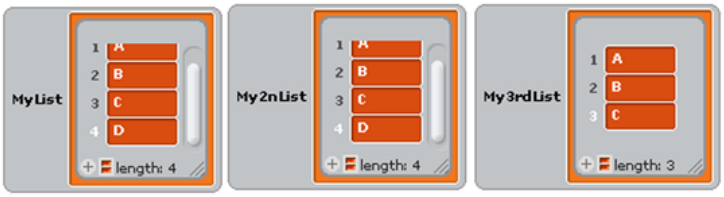
Der Reporter `List` liefert eine neue Liste. Über die kleinen schwarzen Pfeile kann eingestellt werden, wie viele Elemente die Liste haben haben soll. Es ist auch möglich, eine leere Liste zu erzeugen oder bei Entstehung gleich Werte in die Liste einzutragen.

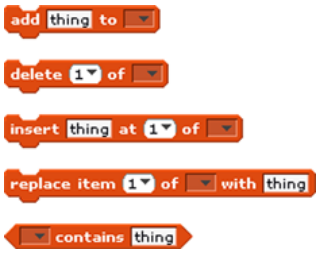
Der Reporter `item(1)of(Liste)` liefert ein Element der Liste. Voreingestellt für den ersten Parameter sind 1, *last* und *any*. Dabei bedeutet *any* ein zufälliges Element der Liste, nicht alle Elemente. Für den ersten Parameter können auch Variablen eingesetzt werden.

Die Länge einer Liste, also die Anzahl an Elementen, lässt sich in BYOB und SNAP! durch den dunkelroten Befehl `length of(list)` bestimmen. Der grüne Befehl `length~of(Text)` dient zur Bestimmung der Länge einer Zeichenkette, also beispielsweise zur Längenbestimmung eines Elements der Liste. BYOB liefert bei falscher Benutzung falsche Zahlenwerte. SNAP! hingegen gibt bei falscher Benutzung des dunkelroten Befehls ein „Error“aus. Bei falscher Benutzung des grünen `length~of(Text)`−Befehls gibt SNAP! trotzdem den richtigen Wert aus.

Der Reporter `copy~of(Liste)` liefert eine eigenständige Kopie einer Liste. Beim Kopieren von Listen muss darauf geachtet werden, dass die Funktion `copy~of(Liste)` genutzt wird. Sonst wird lediglich die Speicheradresse kopiert und Änderungen an der Ursprungsliste werden auch in der Kopie durchgeführt. Das folgende Skript demonstriert die Unterschiede:

Beim Ausführen dieses Scripts kann man sehen, dass der Nachtrag nicht nur in MyList, sondern auch in MyList1 zu sehen ist, da hier auf den COPY-Befehl verzichtet wurde:





Der Commandblock `add(thing)to(Liste)` fügt eine neues Element hinten an die gegebene Liste an. **Achtung, häufigster Schülerfehler: Der Befehl `add` ist in BYOB/SNAP! nicht zur Addition von Zahlen geeignet!**

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 *any*. Dabei bedeutet *any* ein zufälliges Element der Liste, nicht alle Elemente. 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 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 *any*. Dabei bedeutet *any* ein zufälliges Element der Liste, nicht alle Elemente! 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 *any*. Dabei bedeutet *any* ein zufälliges Element der Liste, nicht alle Elemente. 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.

In Struktogrammen (z.B. Wikipedia oder Abituraufgaben bis 2011) werden Listenelemente häufig mit dem Namen der Liste und einem nachgestellten Index in eckigen Klammern gekennzeichnet:

ZugriffAufListenelemente
minimum ← Matrix [zeile, spalte]
Liste[i] ← Zufallsszahl von 1 bis 100

In der ersten Zeile wird der Variable „minimum“ das Element zugewiesen, das unter „Zeile/Spalte“ in der Matrix steht:

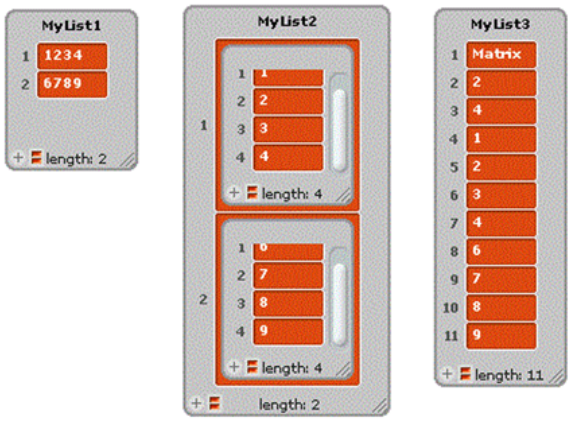
`set minimum to item(spalte)of(item(zeile)of(Matrix)).`

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).`

4.6 Zweidimensionale Reihung[4]

Es gibt verschiedene Arten, eine zweidimensionale Reihung (Matrix oder Tabelle) in BYOB zu implementieren.



Vor- und Nachteile:

Typ	Vorteile	Nachteile
lineare Liste	einfache Handhabung, sehr gut geeignet für kleine Reihungen	für grosse Reihungen sehr unübersichtlich
Liste mit Zeichenketten	schnell, gut geeignet, wenn die Tabelle nur gelesen wird	kein Block zum Austausch eines Elements, ohne Trennzeichen nur nur ein Zeichen pro Zeile
verschachtelte Liste	leichter Zugriff, gute Handhabung	kein händischer Import/Export

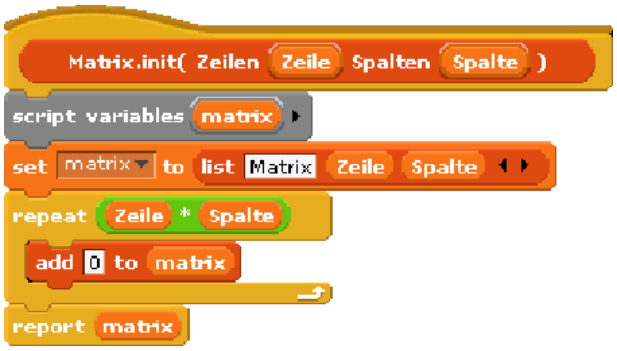
4.6.1 lineare Liste

In einer einfachen Liste werden die Zeilen der Liste nacheinander geschrieben. Bei einer Liste mit z.B. 5 Spalten gehören die ersten 5 Elemente zur ersten Zeile, die nächsten 5 zur zweiten usw. Treten in einem Programm Matrizen verschiedener Größe auf, dann kann man dem Rechnung tragen, indem z.B. im ersten Listenelement das Wort Matrix steht, im zweiten die Anzahl der Zeilen und im dritten die Anzahl der Spalten. In diesem Fall ergibt sich für den Zugriff auf die einzelnen Listenelemente die Formel

$$(Zeile - 1) \cdot Breite + Spalte + 3$$

Der Block `matrix.init()` erstellt eine solche Liste, indem er die Gesamtanzahl der Zellen der Liste als Einzelelemente, welche eine 0 enthalten, in eine Script-Variable schreiben und anschließend zurückgibt.

[4] Mirco Troue



Der Block `matrix.get()` gibt das gesuchte Element aus. Dazu kann man dem Block die Parameter Zeile und Spalte übergeben.

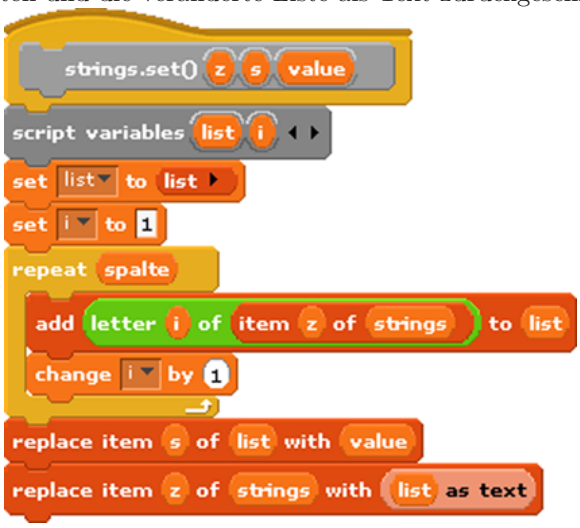
Der Block `matrix.set()` gibt eine Möglichkeit an, die Zellen der Liste zu editieren. Als Parameter dient zusätzlich der Wert (value) für den neuen Inhalt. Die Zelle der Zeile z und Spalte s wird mit dem Inhalt value überschrieben.

4.6.2 Liste mit Zeichenketten

Ebenso lassen sich Zeilen als Einzelelemente darstellen, welche eine Zeichenkette enthalten. Die erste Zeile entspricht dem ersten Element usw. Der Init-Block schreibt zunächst eine ganze Zeile mit „spalte“ Elementen. Diese wird anschließend als Element zur Liste „list“ hinzugefügt und ausgegeben.

Der GET-Befehl ist in diesem Fall sehr kurz, da lediglich der s-te Buchstabe des z-ten Elements ausgegeben werden muss. Auch hier werden als Parameter wieder die Zeile z und die Spalte s übergeben.

Das Bearbeiten der Zeichenketten lässt sich realisieren, indem das entsprechende Element der zu bearbeitenden Zeile in eine Liste gespeichert wird. Jedes Element entspricht nun wieder einer Spalte. Das passende Element wird bearbeiten und die veränderte Liste als Text zurückgeschrieben.



4.6.3 Verschachtelte Liste

Die dritte Möglichkeit eine Reihung zu gestalten besteht darin, in einzelne Elemente einer Liste, welche für die Zeilen stehen, weitere Elemente für die Spalten einzutragen. Somit entsteht eine „Liste in einer Liste“.

Die Prozedur zum Erstellen einer solchen Liste erstellt eine Liste für die einzelnen Spalten. Diese Spaltenliste wird nun als Element in die Zeilenliste eingetragen, welche ausgeben wird.

Ein Element kann ausgelesen werden, indem auf die Elemente in einem Element zurückgegriffen wird.

Analog dazu verläuft das Setzen neuer Werte, da hier ebenso einfach Elemente in Elementen verändert werden können.

Kapitel 5

Grundlagen der Programmierung

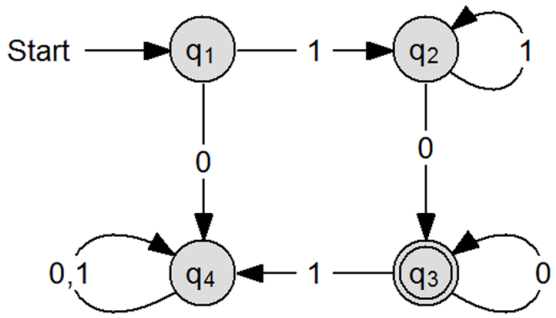
5.1 Programmierung von endlichen Automaten

5.1.1 Funktionsweise von endlichen Automaten

Ein deterministischer endlicher Automat (DEA) besteht aus

1. einer endlichen Menge von Zuständen Q,
2. einem Eingabealphabet Σ,
3. einer Überführungsfunktion δ, die zu Zustand und Eingabezeichen den Folgezustand berechnet,
4. einem Startzustand (hier „q1“),
5. einer endlichen Menge von akzeptierenden Zuständen E (Der häufig verwendete Begriff „Endzustände“ ist mißverständlich, weil der Automat in jedem Zustand enden kann).

Ein DEA dient zur Überprüfung einer Folge von Eingabezeichen, die gemeinsam ein Wort bilden. Der DEA überprüft, ob das jeweils gegebene Wort dazu führt, dass der DEA vom Startzustand in einen akzeptierenden Zustand (Endzustand) übergeht oder nicht. Alle Worte, also alle Folgen von Eingabezeichen, die in einem akzeptierenden Zustand enden, gehören zur Sprache des DEA. Der DEA beginnt im Startzustand. Er geht dann das Wort Zeichen für Zeichen durch und berechnet mit Hilfe der Überführungsfunktion jeweils aus aktuellem Zustand und aktuellem Eingabezeichen den Folgezustand. Beispiel:



Dieser DEA akzeptiert alle Worte, die mit mindestens einer Eins beginnen, auf die dann eine Anzahl von Nullen folgt (mindestens eine). Alle anderen Worte führen ab einer bestimmten Länge in den Fehlerzustand q4. Hinweis: Wir nummerieren die Zustände hier mit q1 beginnend durch (statt mit q0 zu beginnen), um uns die Arbeit in den weiteren Schritten zu erleichtern.

5.1.2 Erste Version des DEA

Für die Überführungsfunktion verwenden wir einen Reporter Delta (diesen Buchstaben gebraucht AutoEdit) mit den Parametern Zustand und EZ (Eingabezeichen). Delta liefert den Folgezustand. Man sieht, dass Delta überwiegend aus geschachtelten Verzweigungen besteht. Zunächst müssen die vier Zustände unterschieden werden und dann das jeweilige Eingabezeichen. Da wir hier nur zwei Eingabezeichen haben, reicht zur Unterscheidung der Eingabezeichen jeweils eine IF-ELSE-Schleife aus. Bei drei und mehr Eingabezeichen müsste für jedes Eingabezeichen eine eigene IF-Schleife eingesetzt werden, z.B.

```
if Zustand = 0
    if EZ = 0
       set Folgezustand to ...
    if EZ = 1
       set Folgezustand to ...
    if EZ = 2
       set Folgezustand to ...
```

Nur beim Zustand q4 vereinfacht sich die Funktion, weil dort alle Eingaben wieder nach q4 führen. Eine Alternative bestünde darin, dass überall dort, wo der Folgezustand gesetzt wird `(set Folgezustand to(Wert))`, der Folgestand mit `report` gleich zurückgegeben wird.

Für die Eingabe verwenden wir ebenfalls einen Reporter. Das scheint jetzt etwas überdimensioniert, aber wir wollen in einem zweiten Schritt ungültige Eingaben herausfiltern und lagern die Eingabe deshalb aus:

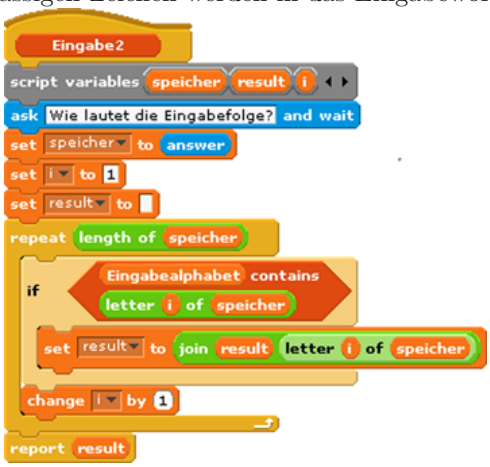
Für unseren DEA benötigen wir nun noch drei globale Variable: den Zustand, das Wort und eine Zählvariable i, weil wir das Wort zeichenweise durchgehen müssen. Das Hauptprogramm setzt den Zustand auf 1 (Startzustand), i ebenfalls auf 1 und das Wort auf die eingegebene Zeichenfolge. Dann wird für jeden Buchstaben `(letter(i)of(wort))` aus Eingabezeichen und Zustand der Folgezustand bestimmt. Zum Schluss überprüft das Programm, ob der akzeptierende Zustand 3 erreicht wurde und macht eine entsprechende Ausgabe:

5.1.3 Erste Verbesserung: Filterung der Eingabe

In der jetzigen Fassung ist der Automat nicht gegen Fehleingaben gesichert. Z.B. wird das Wort „120“ akzeptiert, obwohl 2 nicht zu den gültigen Eingabezeichen gehört. Nun könnte man in die Eingabe eine Abfrage einbauen, ob das Wort nur

aus Einsen und Nullen besteht. Wir wählen hier einen anderen Weg, der sich stärker an der Definition des DEA orientiert und erstellen eine neue Variable Eingabealphabet, in der wir die Eingabezeichen als Liste speichern:

Innerhalb unseres Reporters Eingabe gehen wir die Zeichenfolge einzeln durch und überprüfen, ob die eingegebenen Buchstaben zum Eingabealphabet gehören. Dafür gibt es den Block `(Liste) contains (thing)`. Nur die zulässigen Zeichen werden in das Eingabewort aufgenommen.



Wenn ein anderes Eingabealphabet vorliegt, muss lediglich die Liste „Eingabealphabet“ entsprechend ergänzt werden.

5.1.4 Überführungsfunktion als Tabelle

Schon für unseren kleinen DEA wird die Überführungsfunktion Delta mit all den Fallunterscheidungen unübersichtlich. Das gilt erst recht, wenn wir mehr Zustände oder mehr Eingabezeichen haben. Eine viel übersichtlichere Darstellung der Überführungsfunktion findet sich in AutoEdit als Tabelle:

δ	0	1
q_1	q_4	q_2
q_2	q_3	q_2
q_3	q_3	q_4
q_4	q_4	q_4

Eine mögliche Lösung für Delta wäre also: Suche mit Hilfe des aktuellen Zustands die richtige Zeile und mit Hilfe des Eingabezeichens die richtige Spalte und gib den Zustand im Schnittpunkt von Zeile und Spalte als neuen Zustand aus.

An dieser Stelle wird deutlich, warum wir die Zustände mit Eins beginnend durchnummeriert haben. Wenn wir auf das Q verzichten, also die Zustände nur mit 1, 2, 3, 4 bezeichnen, dann gibt der Zustand direkt die Zeile an. Bei den Eingabezeichen brauchen wir bei diesem Eingabealphabet nur 1 addieren, um die richtige Spalte zu verhalten.

Um die Tabelle der Überführungsfunktion übersichtlich zu halten, speichern

wir sie als Liste, wobei wir die Zustände mit 1, 2, 3, 4 bezeichnen und jede Zeile als ein Wort speichern. Dazu fügen wir im Hauptprogramm folgende Zeile ein:

Mit Hilfe der globalen Variablen Tabelle lässt sich Delta dann vereinfachen:

Diese Version des DEA lässt sich problemlos an einen anderen endlichen Automaten anpassen. Wir müssen lediglich das Eingabealphabet verändern, die Tabelle für die neue Überführungsfunktion einsetzen und ggf. die Abfrage der akzeptierenden Zustände ändern. Wenn das Eingabealphabet sich nicht mehr so leicht in Spalten umrechnen lässt (z.B. wenn Buchstaben vorkommen), brauchen wir noch einen Reporter, der zu jedem Eingabezeichen die richtige Spalte liefert.

5.2 Rund um den MOD-Befehl[1]

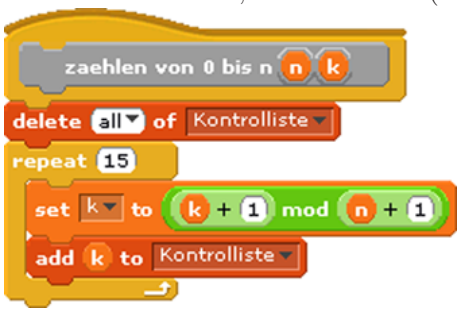
Ein wichtiger Befehl bei der Programmierung ist der MOD-Befehl. `a Mod b` liefert den ganzzahligen Rest bei der Division von a durch b. Beispiel: 13465 mod 100 = 65. Als Reste können alle natürlichen Zahlen auftreten, die kleiner sind als der Divisor, in unserem Beispiel alle Zahlen von 0 bis 99.

5.2.1 Zählen mit dem MOD-Befehl

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

a) Zählen von 0 bis n

Bei der ganzzahligen Division durch n treten als Reste die Zahlen von 0 bis (n-1) auf. Wenn wir zum Zählen aber die Zahlen von 0 bis n benötigen, müssen wir nicht durch n teilen, sondern durch (n+1):



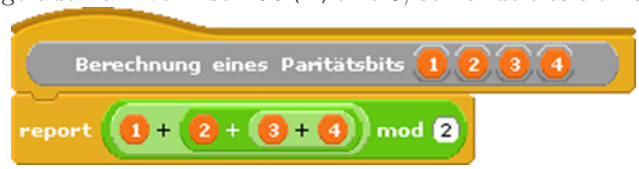
b) Zählen von 1 bis n

Die Division durch n liefert die Reste von 0 bis (n-1). Wennn wir die Zahlen von 1 bis n benötigen, müssen wir jeweils nur 1 hinzuaddieren, also (k mod n) +1

[1] Daniel Marx

5.2.2 Berechnung eines Paritätsbits

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. Beispiele:
1001 enthält zwei Einsen, also wird eine Null angefügt: 10010
0111 enthält drei Einsen, also wird eine Eins angefügt: 01111
Ob eine Zahl gerade oder ungerade ist, in diesem Fall wird so ergänzt, dass es am Ende eine gerade Anzahl an Einsen gibt. Ist `mod(2)` eine 1, so liegt eine ungerade Zahl vor. Ist `mod(2)` eine 0, so handelt es sich um eine gerade Zahl.



5.2.3 DIV (ganzzahlige Division)

Mit (Nenner mod Teiler) wird errechnet, welcher Rest übrigbleiben wird. Wird dieser Rest vom Nenner abgezogen, ist der übrigbleibende Teil durch den Teiler ohne Rest teilbar.

5.2.4 Dezimalzahl in Dualzahl umwandeln

Bleibt bei der ganzzahligen Division durch 2 der Rest 1 über, so wird eine 1 in das Ergebnis geschrieben. Dies wird mit dem Ergebnis der ganzzahligen Division wiederholt, dabei wird der neue Rest an die Stelle vor den bis hierhin schon ermittelten Teil der Dualzahl geschrieben. Sobald die ganzzahlige Division 0 ergibt, ist die Dualzahl fertig.

5.2.5 Euklidischer Algorithmus für den größten gemeinsamen Teiler (ggT)

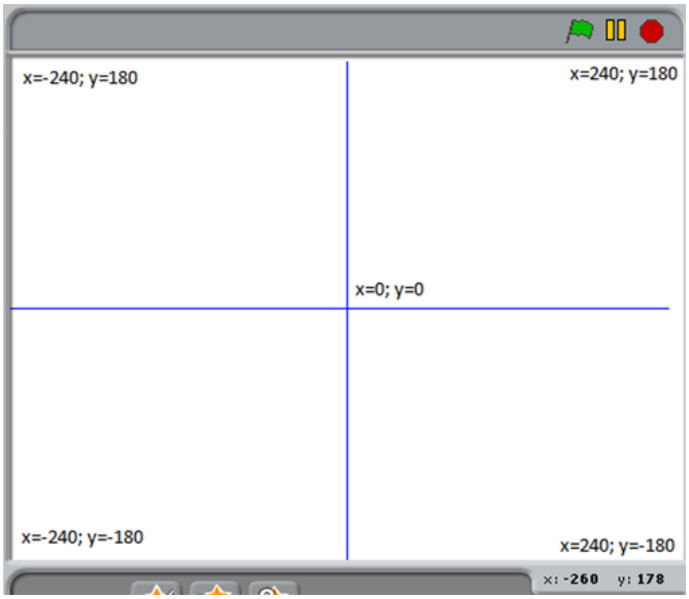
Für alle positiven ganzen Zahlen a, b, k mit a >b gilt: Wenn k beide Zahlen a und b teilt, dann auch die Differenz a − b.

Euklidischer Algorithmus zur Berechnung des ggT :
Gegeben sind zwei natürliche Zahlen a und b (a>b).
1. Solange a $\neq$ 0, wiederhole 2. und 3.
2. Falls b>a, vertausche die Zahlen a und b.
3. Ersetze a durch a mod b.
4. Der größte gemeinsame Teiler ist die Zahl b.

5.3 Zeichnen von Messreihen

Beim Zeichnen von Graphen allgemein ist das Problem zu lösen, dass ein bestimmter mathematischer Koordinatenbereich auf einem bestimmten Bildschirmausschnitt dargestellt werden soll.

Zunächst ist also die Frage zu klären, welcher Bildschirmausschnitt benutzt werden soll. Zur besseren Unterscheidung benutzen wir für die mathematischen Koordinaten den Index m, für die Bildschirmkoordinaten den Index b.



BYOB kennt x-Werte zwischen -240 und +240 und y-Werte zwischen -180 und +180. In diesem Bereich können wir das $x_b$min, $x_b$max, $y_b$min und $y_b$max festlegen.

Zweitens ist zu klären, welcher mathematische Bereich dargestellt werden soll. Bei Meßreihen beginnen in der Regel sowohl Zeit als auch Meßwerte bei Null. Es gilt also $x_m\text{min} = y_m\text{min} = 0$. Der maximale x-Wert $x_m$max richtet sich nach der Zeitdauer der Messung, der maximale y-Wert $y_m$max nach dem größten Messwert.

Wir benötigen also einen Block XmToXb zur Umrechnung der mathematischen x-Koordinaten in die Bildschirm-x-Koordinaten und einen Block YmToYb zur Umrechnung der mathematischen y-Koordinaten in die Bildschirm-y-Koordinaten. [2]

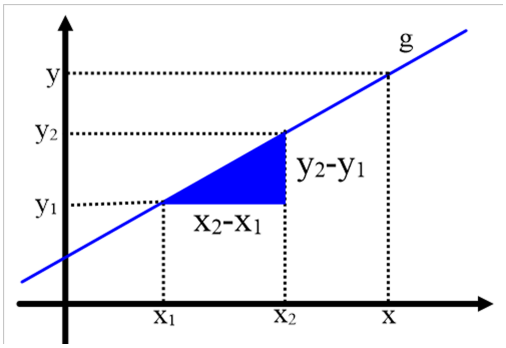
Ein nützliches Werkzeug dafür ist die Zwei-Punkte-Form der Geradengleichung:

$$y - y_1 = \frac{y_2 - y_1}{x_2 - x_1} \cdot (x - x_1)$$

Diese Gleichung kann man wie folgt herleiten:

Werden zwei beliebige Punkte auf einer Geraden gewählt, so ergibt sich immer die gleiche Steigung:

[2] Wenn wir das Programm so erweitern wollen, dass man mit der Maus in die Zeichnung hineinzoomen kann, werden auch Blöcke zur Umrechung der Bildschirm-Koordinaten in mathematische Koordinaten benötigt.

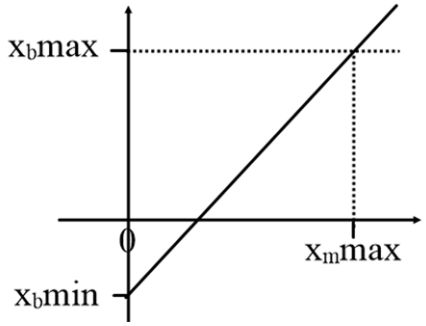


Daraus folgt

$$\frac{y - y_1}{x - x_1} = \frac{y_2 - y_1}{x_2 - x_1}$$

Die Multiplikation mit $(x - x_1)$ liefert dann die Zwei-Punkte-Form der Geradengleichung.

Für die Umrechnung der mathematischen in die Bildschirmkoordinaten gilt:



Die gesuchte Gerade geht also durch die Punkte $(0|x_bmin)$ und $(x_mmax|x_bmax)$.

Beispiel: Nehmen wir an, dass wir einen Bildschirmrand von je 20 Bildpunkten freilassen wollen, um Überschriften und Koordinatenachsen zu ergänzen. Dann ergibt sich $x_bmin = -220$, $x_bmax = +220$, $y_bmin = -160$ und $y_bmax = +160$. Wenn wir jetzt 10 Messwerte darstellen ($x_mmax = 10$) und der größte Messwert 95,8 ist (aufgerundet $y_mmax = 100$), dann ergibt sich folgende Formel zur Umrechnung der x-Koordinaten:

$$y - x_bmin = \frac{x_bmax - x_bmin}{x_mmax - 0} \cdot (x - 0)$$

$$y - (-220) = \frac{220 - (-220)}{10 - 0} \cdot (x - 0)$$

aufgelöst nach y ergibt sich für die Funktion XmToXb:

$$y = 44 \cdot x - 220$$

Entsprechend berechnet man die Funktion zum Umrechnen der y-Koordinaten.

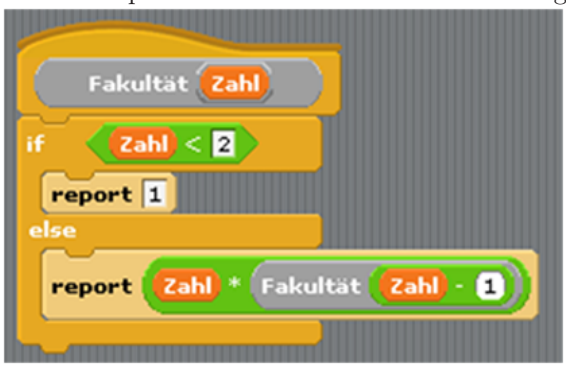
5.4 Rekursion[3]

Unter Rekursion versteht man in der Informatik, daß ein Programm sich selbst aufruft. Dabei muss der Programmierer darauf achten, daß keine Endlosschleife entsteht. Die würde das Programm abstürzen lassen. Die grundlegende Struktur eines rekursiven Blocks ist also eine Verzweigung:

Ohne Abbruchbedingung ist ein rekursiver Block nicht lauffähig.

Häufig müssen wir in der Informatik komplexe Vorgänge implementieren, in denen wiederkehrende Strukturen auftauchen. Wenn man z.B. die Fakultät einer Zahl bestimmt, muss man die Zahl mit allen kleineren natürlichen Zahlen bis zur Eins multiplizieren: $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$

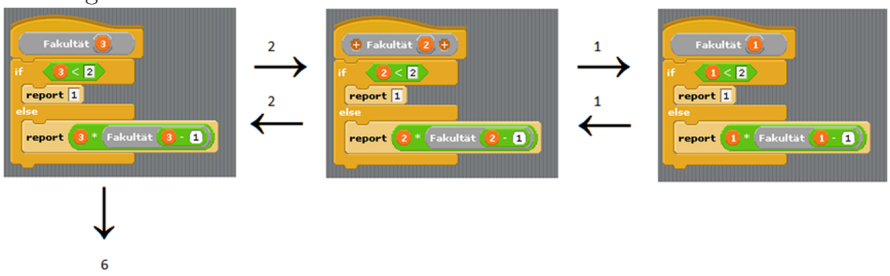
Um lange Codeblöcke mit sich wiederholenden Zeilen zu sparen, nutzen wir das Prinzip der Rekursion. Hierfür erstellen wir einen Reporter-Block, der eine Zahl mit dem Produkt der restlichen kleineren Zahlen multipliziert. Die Bestimmung dieses Produktes gibt der Block an eine andere Instanz von sich selbst weiter, die wieder jeweils nur eine Zahl mit dem noch zu bestimmenden Rest-Produkt multipliziert und die Restarbeit weiterdirigiert.



Sobald nur noch die Eins übrig geblieben ist, darf kein erneuter Aufruf des Fakultät-Blocks geschehen, da die Null bei der Bestimmung der Fakultät nicht berücksichtigt wird. Um also einen erneuten Aufruf zu verhindern, müssen wir eine Abbruchbedingung einbauen. In diesem Fall wird mit „If-Else“ geklärt, ob die nächste Zahl in der Reihe kleiner als Zwei ist. Wenn dem so ist, bleibt nur noch die Eins übrig und kann an den nächsthöheren Block weitergegeben werden.

[3]Niklas Euler

Letztendlich rollt der Fakultät-Block das Problem also von hinten auf: Zunächst wird die Bestimmung des Rest-Produktes immer weiter nach unten in der Blockstruktur gegeben, bis der unterste Fakultät-Block die Eins an den nächst höheren Block meldet, welcher dann einen Teil des Restproduktes bestimmen kann. Dieses kann dann erneut weitergegeben werden usw., bis der oberste Block das Endergebnis ermitteln kann. Die Grafik zeigt beispielhaft die Berechnung von 3!:



Wir unterscheiden verschiedene Arten von Rekursion:

- bei der **linearen Rekursion** wird die Funktion nur einmal aufgerufen (Beispiel: Fakultät)
- bei der **kaskadenförmigen Rekursion** wird die Funktion zwei- oder mehrmals aufgerufen (Beispiel: Berechnung des Binomialkoeffizienten)
- bei der **wechselseitigen Rekursion** rufen sich zwei Funktionen gegenseitig auf.

Vor- und Nachteile der Rekursion:

\+ kompakter Programmcode

\+ (häufig) einfachere Lösungen

\- (häufig) langsamer

\- höherer Speicherbedarf (vor jedem neuen Aufruf müssen alle Zwischenwerte gespeichert werden)

\- fehleranfälliger (Gefahr der Endlosschleife)

Eine große Bedeutung hat die Rekursion bei der Behandlung von Bäumen und Graphen sowie bei vielen Grafiken (Fraktalen).

Kapitel 6

Abstrakte Datentypen (ADTs)

6.1 Grundlagen

Ein abstrakter Datentyp (ADT) ist ein Verbund von Daten und Operationen, die auf diese Daten zugreifen. Die Daten sind „gekapselt“, d.h. ein Benutzer darf nur über die Operationen auf die Daten zugreifen, die in der Definition des ADT festgelegt sind. Auch wenn zur Implementierung eines ADT eine dynamische indizierte Liste (also unsere normale BYOB-Liste) verwendet wird, darf man z.B. nicht über den Index auf einzelne Elemente zugreifen, wenn die Liste der Operationen keinen indizierten Zugriff vorsieht.

Abstrakte Datentypen als Programmierkonzept wurden eingeführt, um einfacher arbeitsteilig zu programmieren und die Wiederverwendbarkeit des Quelltextes zu erhöhen.

Die Thematischen Hinweise für das Zentralabitur kennen zwei Arten von Aufgaben im Zusammenhang mit ADTs:

1. die Implementierung einzelner oder aller Operationen eines ADT
2. die Anwendung eines Abstrakten Datentyps (ADT)

Klassische objektorientierte Programmiersprachen wie Delphi unterscheiden strikt zwischen verschiedenen Datentypen. Das hängt damit zusammen, dass sie in einer Zeit entstanden sind, als Speicherplatz knapp war. Die Zahl 120 lässt sich als 00110001—00110010—00110000 (dezimal 49—50—48) speichern. Das wäre eine Speicherung des ASCII-Codes der drei Ziffern 1, 2 und 0. Andererseits lässt sich 120 = 64+32+16+8 auch als Dualzahl 01111000 speichern. Das spart zwei Drittel des Speicherplatzes ein. Für jede Variable oder Funktion muss ein Delphi-Programmierer angeben, welcher Datentyp jeweils vorliegt, also ob ein STRING, eine INTEGER- oder eine REAL-Zahl vorliegt usw. BYOB/SNAP! hat kein so starres Datenkonzept. Die Zahlen werden ziffernweise gespeichert, so dass man sowohl mit ihnen rechnen kann als auch z.B. mit `letter(i)of(text)` auf einzelne Ziffern zugreifen kann. Diese Flexibilität erkauft man mit einer höheren Verantwortung des Programmiers, der sich nicht mehr darauf verlassen kann, dass das Programm falsche Typzuweisungen mit

einer Fehlermeldung quittiert. In BYOB/SNAP! sind alle Daten first-class, lassen sich also einer Variable zuweisen oder als Parameter einsetzen, es sei denn, bei der Definition des Parameters wurden Einschränkungen vorgenommen.

Für die Implementierung einer Operation eines ADTs in BYOB/SNAP! gilt also

1. An die Stelle einer Deklaration der Typen tritt eine kurze Beschreibung, in welcher Weise die Datenstruktur implementiert wird, z.B. ob eine lineare Liste, eine Liste von Zeichenketten oder eine verschachtelte Liste für eine zweidimensionale Reihung verwendet wird oder welches Attribut im 1., 2., usw. Listenelement gespeichert wird.

2. Die erste Operation ist jeweils der Konstruktor, der ein entsprechendes Objekt erzeugt. Teilweise findet man für den Konstruktor die Bezeichnung `create`. In Niedersachsen wird in den letzten Jahren statt dessen durchgehend die Bezeichnung des ADT verwendet, also `Schlange(), Stapel(), Baum()`. Als Konstruktor verwenden wir immer einen Reporter. Beim Aufruf braucht der Konstruktur immer einen vorangestellten Namen, damit man das neu erzeugte Objekt auch ansprechen kann, also z.B. `Hilfschlange.Schlange(), My1stStapel.Stapel()`.

3. Alle Operationen, die auf „: Wahrheitswert“ enden, werden als Prädikat implementiert. Ein Prädikat kann nie allein in einer Zeile stehen, sondern nur als Bedingung in einen anderen Block eingesetzt werden.

4. Alle Operationen, die auf „: IrgendeinAndererTyp“ enden, werden als Reporter implementiert. Ein Reporter kann nie allein in einer Zeile stehen, sondern nur als Teil eines anderen Blocks, z.B. als rechter Teil einer Wertzuweisung oder als Anzahl in einer Wiederholungsschleife.

5. Alle übrigen Operationen werden als Command implementiert.

6. Die in Klammern aufgeführten Parameter werden in Klammern in den Kopf der Operation übernommen.

7. Typbezeichnungen für Variable und Operationen gibt es in BYOB/SNAP! nicht. Sie sind deshalb wegzulassen.
   Einzige Ausnahme: Wenn der Typ eines Parameter im Blockeditor eingestellt wurde (z.B. als `number` bei einem Block `Maximum`), wird dies in BYOB/SNAP! hinter einem Doppelpunkt angegeben: `Reporter Maximum (Zahl1, Zahl2: Number)`. Die Angabe des Typs erscheint nur in der Kopfzeile. Sie kann nicht als Parameter oder Variable verwendet werden. Wird ein Block innerhalb eines anderen Skripts verwendet, so wird kein Typ angegeben.

8. Alle Operationen mit Ausnahme des Konstruktors erhalten als ersten Parameter den Namen des Objekts, zu dem die Operation gehört, also `(Schlange)anhaengen(inhalt)` oder noch besser `Schlange.anhaengen(inhalt)`. Der Konstruktor erhält als ersten Parameter seinen Namen, damit man auf das erzeugte Objekt auch zugreifen kann. In einem Programm gibt es in der Regel mehrere Schlangen, Stapel oder Bäume. Der erste Parameter gibt deshalb an, mit welchem Objekt die Operation jeweils ausgeführt wird.

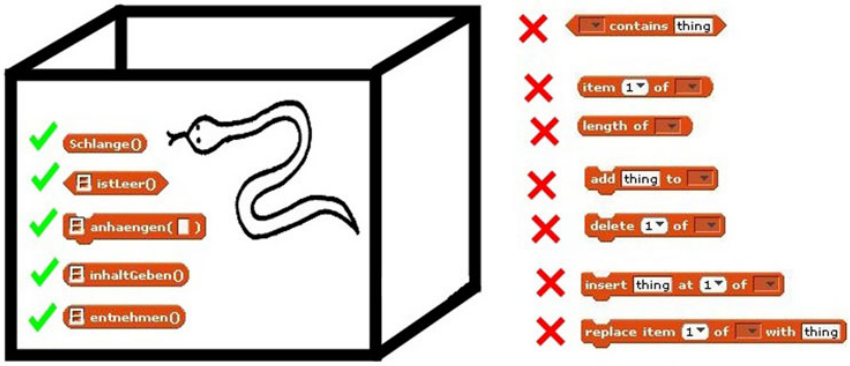
Die zweite Art der Aufgabenstellung ist die Anwendung von Abstrakten Datentypen. Bei der Anwendung von ADTs verwenden wir die Schreibweise `Objektname.Methode`, also

- `MySchlange.anhaengen(inhalt)`
- `Set Ergebnis to BaumA.rechterTeilbaum()`
- `Schlange1.anhaengen(Schlange2.entnehmen)`
- ```
  repeat until StapelA.istLeer()
      StapelB.ablegen(StapelA.entnehmen)
  ```

Wenn neue Operationen implementiert werden sollen, dürfen zum Zugriff auf die Attribute (Inhalte) nur die bereits definierten Methoden verwandt werden. Es ist also nicht zulässig, z.B. bei der Implementierung einer Operation `concat` für zwei Stapel `add item(...)to(...)` oder `delete item(...)of(...)` zu verwenden.

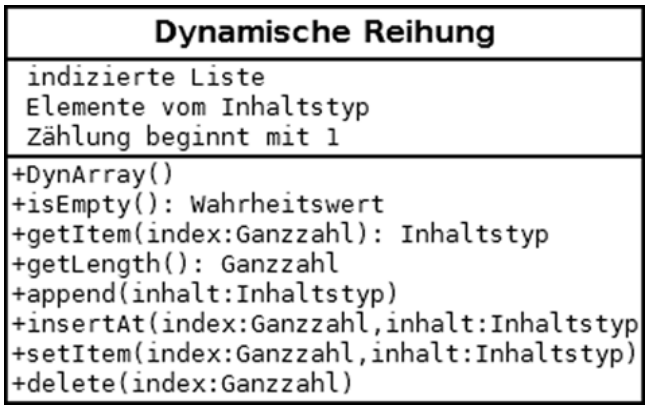
Für die Blöcke gelten dabei die üblichen Regeln: Prädikat-Blöcke können nur für Bedingungen eingesetzt werden. Reporter liefern einen Wert, sie können also nur in Wertzuweisungen oder als Parameter eingesetzt werden. Nur Command-Blöcke dürfen allein in einer Zeile stehen.

Bitte denken Sie daran, dass für ADTs das Prinzip der Datenkapselung gilt. Der häufigste Fehler ist, dass man eine bestimmte Art der Implementierung unterstellt und für den Zugriff die BYOB-Listenoperationen benutzt. Am Beispiel des ADT Schlange zeigt die folgende Grafik, welche Operationen man anwenden darf (grüner Haken) und welche nicht (rotes Verbotskreuz).



6.2 ADT Dynamische Reihung (DynArray)

Für die Abiturprüfung 2018 wird an die Stelle des ADT Liste, der bis 2015 zu den verpflichtenden abstrakten Datentypen gehörte, der ADT Dynamische Reihung treten. Auch in der Arbeitsfassung des Kerncurriculums für Informatik in der Sekundarstufe II vom 19.12.2014 ist dieser ADT enthalten. Der neue ADT ist viel stärker als sein Vorgänger nach dem Vorbild der Listen in BYOB modelliert. So beginnt jetzt die Zählung mit 1 (und nicht mehr mit 0 wie bei statischen Arrays). Der Vorteil, den die Benutzer von BYOB bisher hatten, daß sie über den Index auf jedes einzelne Element einer Liste direkt zugreifen können, wird dadurch etwas verringert.
  ```



Für den Konstruktor `DynArray()` verwenden wir einen Reporter, der den Namen der neuen Reihung als Parameter erhält.

Das Prädikat `isEmpty()` gibt es standardmäßig nicht in BYOB, es kann aber mit Hilfe eines Reporters implementiert werden, der ausgibt, ob eine Liste die Länge Null hat.

Für `getItem(index)` gibt es den Reporter `item(i)of(Liste)`.

`getLength()` heißt `length~of`.

`Append(inhalt)` kann mit `add(thing)to(Liste)` realisiert werden.

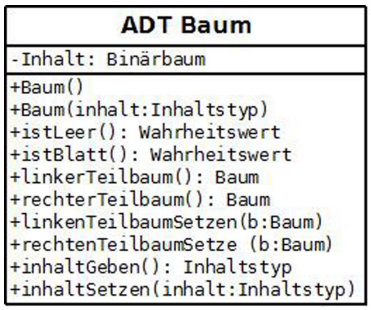
Die Methode `insertAt(index,inhalt)` wurde von `insert(thing)at(i)of(Liste)` übernommen. Beide Operationen funktionieren sowohl innerhalb der Liste als auch an der Stelle direkt hinter dem letzten Listenelement (aber nicht mit anderen Indizes!).

Neu ist `setItem(index, inhalt)`, das dem `replace~item(i)of(Liste)with(thing)` entspricht.

`Delete(index)` hat die gleiche Wirkung beim ADT als auch in BYOB.

6.3 ADT Binärbaum

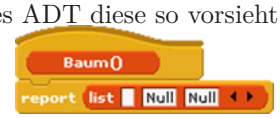
Auch der Binärbaum gehört zu den Abstrakten Datentypen (ADT), die im Zentralabitur vorgegeben sind. Der ADT Binärbaum ist ein schönes Beispiel für die Leistsungsfähigkeit von BYOB/SNAP!. Alle Operationen lassen sich als Einzeiler programmieren, wenn wir den kleinen Trick aus 2.3 anwenden (statt IF ... ELSE ... eine Aussage reporten).



Wir implementieren den Baum mit Hilfe von verketteten Listen. Jeder Knoten besteht dabei aus einer dreielementigen Liste. Das erste Element enthält den

Inhalt des Knotens, das zweite Element den linken Teilbaum und das dritte Element den rechten Teilbaum. Wichtig ist dabei, dass wir auch jedes Blatt als eigene Liste implementieren.

Der Konstruktor `Baum()` liefert eine dreiteilige Liste. Dabei müssen wir darauf achten, dass das erste Element gelöscht wird (sonst enthält es ein Leeerzeichen). In die beiden übrigen Elemente schreiben wir „Null“, weil die Definition des ADT diese so vorsieht.



Der zweite Konstruktor `Baum(inhalt)` unterscheidet sich nur darin, dass der Inhalt jetzt gesetzt wird:

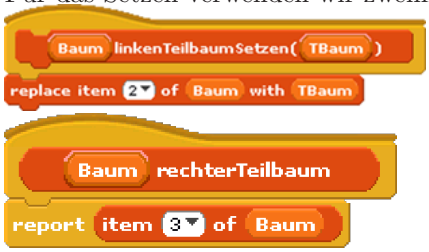
Das Prädikat `istLeer()` prüft, ob der Inhalt leer ist. Wichtig ist dabei, dass das zweite Feld in der Gleichung gelöscht wird:

Das Prädikat `istBlatt()` liefert true, wenn sowohl der linke als auch der rechte Teilbaum Null ergeben.

Der Reporter `linkerTeilbaum()` liefert das zweite Element der Liste:

Der Reporter `rechterTeilbaum()` liefert das dritte Element der Liste:

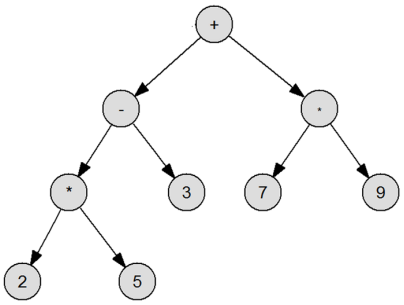
Für das Setzen verwenden wir zweimal den Befehl REPLACE:



InhaltGeben und InhaltSetzen bezieht sich auf das erste Listenelement:

6.4 Ausgabe eines Termbaums

Gegeben sei ein Termbaum, d.h. ein Binärbaum zur Darstellung mathematischer Terme, z.B. der Baum für $(2 \cdot 6) - 3 + (7 \cdot 9)$.



Gesucht ist eine Operation, die den zugehörigen mathematischen Term mit Klammern ausgibt. Dazu nutzen wir die Rekursion. Zunächst stellen wir fest, dass alle Knoten mit Zahlen Blätter des Baumes sind. Knoten, die ein Rechenzeichen enthalten, sind innere Knoten. Halbblätter gibt es in Termbäumen nicht, weil zu jeder Rechenoperation zwei Zahlen gehören. Zur Unterscheidung definieren wir eine Liste Rechenzeichen $= \{+, -, *\}$.

Falls der aktuelle Knoten ein Rechenzeichen enthält, setzen wir eine öffnende Klammer, rufen die Ausgabe des linken Teilbaums auf, fügen das Rechenzeichen hinzu, dann die Ausgabe des rechten Teilbaumes und schließlich die schließende Klammer.

Falls der Knoten eine Zahl enthält, geben wir den Inhalt aus.

Kapitel 7

Häufige Schülerfehler

7.1 Wertzuweisungen

Der häufigste Schülerfehler 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:

Befehle zur Wertzuweisung

Typ	als Variable	als Listenelement
Wahrheitswert		
Zahl	 	
Zeichenkette		

Die Verwendung eines falschen Blocks kann schwerwiegende Folgen haben, weil BYOB keine Prüfung der Typen durchführt.

Beispiel: MyVar habe den Wert „Zahl“. Nun soll eine Eins hinten angefügt werden. Bei einem `change(MyVar) by(1)` geht BYOB davon aus, dass es sich um eine Zahlvariable handelt. Ein etwa vorhandener Text wird gelöscht und dann 1 addiert. MyVar hat also anschließend den Wert 1.

7.2 Verstoß gegen die Datenkapselung

BYOB-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 BYOB-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 BYOB-Listenbefehlen!

7.3 Falsche Verwendung des ADD-Blocks

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!

7.4 Falsche Verwendung von Parametern

Parameter werden beim Aufruf eines Blocks übergeben. Sie stehen im Quelltext in Klammern in der Kopfzeile (Beispiel: `move(10)steps`). Es ist weder notwendig noch sinnvoll, den Wert von Parametern durch eine Benutzerabfrage oder eine Zufallsfunktion festzulegen.

7.5 Alleinstehender Reporter

Methoden zur Anzeige der Eigenschaften von Objekten haben häufig Namen, die mit einem Verb zusammengesetzt sind, wie `inhaltGeben` oder `getLength`. Dennoch handelt es sich um Reporter, die in BYOB 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.entnehmen()` (bis Abitur 2017) bzw. `Schlange.dequeue()` (ab Abitur 2018) kann nicht allein in einer Zeile stehen, wohl aber als Parameter in einer Wertzuweisung z.B.

`Schlange1.anhaengen(Schlange2.entnehmen)` bzw.

`Schlange1.enqueue(Schlange2.dequeue)`.

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

7.6 Kopieren von Listen

Eine gefährliche Fehlerquelle ist es, wenn Listen durch einfache Zuweisung ohne Verwendung des `Copy∼of(List)`-Blocks kopiert werden. In diesem Fall erstellt BYOB 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 in der anderen ebenfalls auf.

7.7 Das unsichtbare Leerzeichen

Der Block `set(Variable)to(Wert)` enthält ein unsichtbares Leerzeichen. Wird eine Zeichenkette buchstabenweise aufgebaut, so muss zunächst die Variable initialisiert werden, indem das Leerzeichen gelöscht wird. Ansonsten enthält die Zeichenkette als ersten Buchstaben ein Leerzeichen.

Kapitel 8

Notation von BYOB/SNAP!

In Klausuren und Prüfungen gibt es die Aufgabenstellung „Implementiere eine Operation“. **Das bedeutet immer, dass Quelltext zu schreiben ist.** Für die handschriftliche Notation von BYOB-Quelltext gelten folgende Regeln:

- Wir verwenden ab Klasse 10 einheitlich die englische Schreibweise für die Blöcke.
- Vor jeden Block steht der entsprechende Typ, also Command, Reporter, Prädikat.
- Variablen werden in der InFixCap-Schreibweise geschrieben, d.h. der erste Buchstabe wird klein und jeder erste Buchstabe eines neuen Wortes wird groß geschrieben. Diese Schreibweise wird auch im Abitur verwendet, z.B. `istLeer()`. In eigenen Variablenbezeichnungen sollte man Leerzeichen möglichst vermeiden.
- Die Leerzeichen zwischen Worten in den Namen von Blöcken werden durch die Tilde ∼ ersetzt.
- Klammern markieren den Beginn und das Ende von Parameterlisten.
- Aufeinanderfolgende Parameter werden durch Komma getrennt.
- Textvariable müssen häufig initialisiert werden, d.h. die Null bzw. das Leerzeichen, das BYOB automatisch einsetzt, muss gelöscht werden. Dies schreibt man mit zwei Anführungszeichen:

  ```
  set MyWord to „"
  ```

- Wenn Parametern im Blockeditor bestimmte Typen zugewiesen wurden, dann werden diese im handschriftlichen Quelltext mit „: Typ“ notiert, z.B.

  ```
  command einfuegen(position: number, inhalt: List)
  ```

  Hier darf bei Position nur eine Zahl eingesetzt werden, bei Inhalt eine Liste.

- Der Inhalt von Listen wird in geschweiften Klammern dargestellt:

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

- Einrückungen im BYOB 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 die Übersicht verloren. **Es wird daher dringend empfohlen, in diesem Fall die jeweilige innere Reichweite der Schleife durch eine Klammer zu kennzeichnen!**

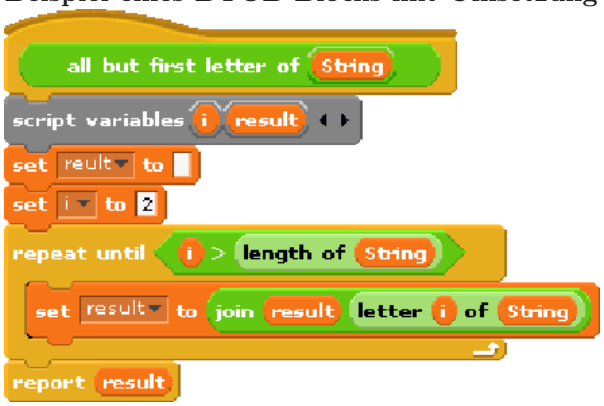
Abweichende Schreibweise bei ADTs / UML

Objekte können zwar als Parameter definiert werden, aber in Anlehnung an die Schreibweise in anderen Programmiersprachen wird der Objektname der Methode vorangestellt und mit Punkt abgesetzt. Leider lässt sich der Punkt in BYOB nicht als Teil des Blocknamens verwenden. Wir weichen bei der Implementierung deshalb an dieser Stelle von der BYOB-Notation ab und verwenden die Schreibweise „Objekt.Methode“.

Beispiele:

Ein Stapel namens MyStapel soll erzeugt werden. Wir schreiben:
`MyStapel.Stapel()` statt `set MyStapel to Stapel()`
Die Zahl 18 soll auf MyStapel abgelegt werden. Wir schreiben:
`MyStapel.ablegen(18)` statt `(MyStapel)ablegen(18)`

Beispiel eines BYOB-Blocks mit Umsetzung in Quelltext:




```
reporter all-but-first-letter-of(String)
script-variables Result, i
set Result to „"
set i to 2
repeat until I > length-of(String)
    set Result to join(Result,letter(i)of(String))
    change i by 1
report Result
```

Kompetenzraster Programmmierung

Level	Anfänger		Fortgeschrittene		Experten	
Thema	A1	A2	B1	B2	C1	C2
Konzepte **(K)**	Ich kann die Arten von Blöcken und deren Verwendung erläutern.	Ich kann Sequenzen und Verzweigungen (ein- und zweiseitig) verwenden und in Struktogrammen zeichnerisch darstellen.	Ich kann globale und lokale Variablen und Parameter erkennen und richtig verwenden.	Ich kann vor- und nachprüfende Schleifen konstruieren und den Unterschied zwischen Abbruch- und Wiederholungsbedingung erläutern.	Ich kann die Funktionen einer Zählschleife nennen und eine Zählschleife konstruieren.	Ich kann das Prinzip der Rekursion (einschließlich der Abbruchbedingung) erläutern und Rekursion anwenden.
Zeichenketten **(Z)**	Ich kann Zeichenketten zusammensetzen.	Ich kann auf einzelne Buchstaben von Zeichenketten zugreifen.	Ich kann erläutern, wie der alphabetische Vergleich in BYOB/Snap! funktioniert.	Ich kann die Häufigkeit von Buchstaben in Zeichenketten bestimmen.	Ich kann Buchstaben in ASCII-Code umwendeln und umgekehrt und mit Buchstaben rechnen.	Ich kann einen Block schreiben, der Buchstaben in einer Zeichenkette ersetzt.
Rechnen **(R)**	Ich kann die Grundrechenarten auf Zahlvariable anwenden.	Ich kann eine Schaltfunktion in einen Block umsetzen.	Ich kann den Abstand zweier Punkte berechnen und eine Zahl auf zwei Nachkommastellen runden.	Ich kann mit dem MOD-Befehl zählen.	Ich kann Reste auswerten und eine Dezimal- in eine Dualzahl umwandeln.	Ich kann die De' Morganschen Gesetze anwenden, um eine Bedingung zu vereinfachen.

Level	Anfänger		Fortgeschrittene		Experten	
Thema	A1	A2	B1	B2	C1	C2
Listen **(L)**	Ich kenne zwei Möglichkeiten, eine Liste anzulegen (nur BYOB).	Ich kann auf einzelne Elemente einer Liste zugreifen.	Ich kann einzelne Elemente an eine Liste anhängen, einfügen und löschen.	Ich kann zwei Listen zu einer Liste zusammenführen.	Ich kann eine Liste sortieren.	Ich kann eine verkettete Liste erstellen und benutzen.
Zweidimensionale Reihungen (Matrizen **M**)	Ich kann 3 Implementierungen für eine zweidimensionale Reihung und deren Vor- und Nachteile erläutern.	Ich kann eine zweidimensionale Reihung mit einem Startwert initialisieren.	Ich kann mit einer geschachtelten Zählscheife auf die Elemente einer zweidimensionalen Reihung zugreifen.	Ich kann Werte wie Summe, Maximum, Mittelwert einer zweidimensionalen Reihung bestimmen.	Ich kann einen Graphen durch eine Adjazenzmatriz oder eine Adjazenzliste beschreiben.	Ich kann Operationen mit Matrizen implementieren (Addition, Matrizenmultiplikation).
Abstrakte Datentypen (ADTs **A**)	Ich kann die Operationen der ADTs Schlange und Stapel anwenden.	Ich kann die Operationen derm ADTs Schlange und Stapel implementieren.	**eN** Ich kann Bäume und Operationen mit Bäumen graphisch darstellen.	**eN** Ich kann den Inhalt von Bäumen alphabetisch sortiert ausgeben.	**eN** Ich kann in Bäumen suchen und Elemente einfügen.	**eN** Ich kann mathematische Operationen mit Bäumen durchführen.

Notizen