Stapel, Schlange, Liste, Baum mit Snap!

Fritz Hasselhorn

1. März 2023

Gymnasium Sulingen, Schmelingstraße 37, 27232 Sulingen

Inhaltsverzeichnis

1. Speicherverwaltung und Typisierung

Das folgende Kapitel gibt eine kurze Einführung in Rechneraufbau, Speicherorganisation und klassische Datentypen. Ziel ist nicht eine Ansammlung historischer Kenntnisse, sondern ein Verständnis für die Relikte aus der Geschichte der Informatik, die wir heute noch vorfinden. Warum lässt sich eine Liste nicht einfach mit einem SET-Befehl kopieren und warum beginnen manche Datentypen die Zählung ihrer Elemente mit Null? Warum tauchen in Abituraufgaben Reihungen und zweidimensionale Reihungen auf, wenn es in Snap! nur Listen gibt?

1.1. Von-Neumann-Architektur

Die ersten modernen Computer wurden im Zweiten Weltkrieg entwickelt. In Bletchley Park in Großbritannien entstanden unter maßgeblicher Mitwirkung von Alan Turing, dem Begründer der Informatik, die Turingbomben, Maschinen zur Entschlüsselung des deutschen Funkverkehrs, der mit der Enigma verschlüsselt wurde.

1941 konstruierte Konrad Zuse in Berlin seinen Z3, einen elektromechanischen Rechner mit 600 Relais im Rechenwerk und 1600 Relais im Speicherwerk.

Der amerikanische Eniac 1945 bestand aus Elektronenröhren. Auch der Mark II verwendete neben Elektronenröhren elektromechanische Relais. Die amerikanische Informatikerin Grace Hooper fand 1947 auf der Suche nach einem Fehler in einem dieser Relais eine Motte und klebte sie in das Logbuch zusammen mit dem Satz „First actual case of bug being found.“ Daher stammt der Ausdruck ”debuggingfür die Fehlersuche in Programmen.

Diese ersten Rechner wurde über Lochstreifen, Lochkarten, Steckfelder und große Schaltertafeln programmiert. War ein Programm abgeschlossen, musste die Maschine von Hand umgebaut und umprogrammiert werden, bevor ein neues Programm gerechnet werden konnte. Im Jahr 1945 veröffentlichte John von Neumann seine Ideen für eine neue Computerarchitektur.

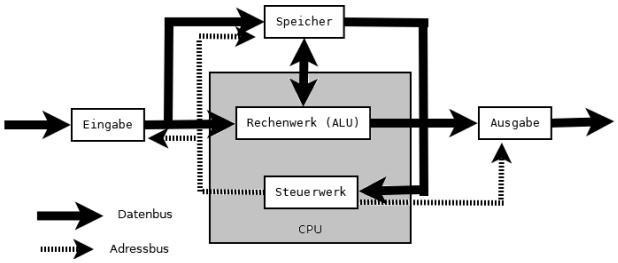


Abbildung 1.1.: Von-Neumann-Rechner

Die Von-Neumann-Architektur unterteilt den Rechner in sechs Funktionseinheiten:

- Steuerwerk
- Rechenwerk
- Speicher
- Eingabesystem
- Ausgabesystem
- Datenbus und Adressbus

Das **Steuerwerk** ist die übergeordnete Einheit des Rechners, es steuert den Ablauf eines Computerprogramms Schritt für Schritt, analysiert und greift dazu über den Steuerbus auf die übrigen Funktionseinheiten zu.

Das **Rechenwerk** ist für arithmetische und logische Operationen zuständig.

Der **Speicher** besteht aus einzelnen Speicherzellen, die ein **Bit** speichern, d.h. eine Null oder eine Eins. Jeweils acht dieser Bits sind zu einem **Byte** zusammengeschaltet. Auf die einzelnen Elemente des Speichers kann über deren Nummer (**Adresse**) zugegriffen werden.

Das **Eingabesystem** besteht bei modernen Rechnern aus Tastatur und Maus sowie weiteren Eingabegeräten wie Mikrofon und Kamera.

Das **Ausgabesystem** besteht aus Monitor und Drucker, bei Rechnern, die Anlagen steuern, auch aus Motoren, Heizspiralen und anderen technischen Geräten.

Der **Datenbus** transportiert die Daten zwischen den verschiedenen Funktionseinheiten. In modernen Rechnern besteht der Datenbus in der Regel aus 64 parallelen Datenleitungen, man spricht deshalb von einem 64-Bit-Rechner. Der **Adressbus** dient zur Ansprache der verschiedenen Speicherstellen über ihre Adresse.

Der entscheidende Fortschritt an von Neumanns Konzept ist, dass der Aufbau des Rechners unabhängig von den zu bearbeitenden Problemen wird. Der zeitintensive Umbau entfällt. Zur Lösung eines Problems muss nur eine Bearbeitungsvorschrift, das Programm, eingegeben und im Speicher abgelegt werden. Ohne ein Programm ist die Maschine nicht arbeitsfähig. Der Speicher enthält Programme, Daten, Zwischen- und Endergebnisse.

Aufeinander folgende Befehle eines Programms werden in aufeinander folgenden Speicherzellen abgelegt. Das Ansprechen des nächsten Befehls geschieht vom Steuerwerk aus durch Erhöhen der Befehlsadresse um Eins. Ein Von-Neumann-Rechner muss arithmetische Befehle wie Addieren und Multiplizieren ausführen können, aber auch logische Befehle wie Vergleiche. Alle Daten (Befehle, Adressen, Eingaben, Ausgaben usw.) werden als Dualzahlen codiert.

1.2. Einfache Datentypen

Der Speicher teilt sich in zwei Bereiche auf, den Stack (Stapel oder Keller) und den Heap (Haufen).

Der Stack funktioniert nach den LIFO-Prinzip, d.h. *last in first out*, d.h., die Daten, die zuletzt abgespeichert wurden, werden als erste wieder verarbeitet. Variable werden im Stack abgelegt. Für die Variablen werden in den klassischen Programmiersprachen, z.B. Delphi, symbolische Namen vergeben. Delphi führt eine Art Adressbuch, in dem die Zuordnung von Variablennamen zu Speicheradressen vermerkt ist.

Die einfachen Datentypen, also Zahlen, Buchstaben und Wahrheitswerte werden direkt in den Stack geschrieben. Bei Programmausführung werden die Namen dann durch die Werte an der entsprechenden Speicheradresse ersetzt.

Beispiel (Delphi):

VAR x, y, vx, vy: integer; Geschlecht: char; berufstaetig: boolean; Alter: integer; Name, Vorname, Beruf: string;

x := 123; y := 87; vx := -5; vy := 3; Geschlecht := 'w'; berufstaetig := true; Alter := 23; Beruf := 'Informatikerin'; Name := 'May'; Vorname := 'Hannah';

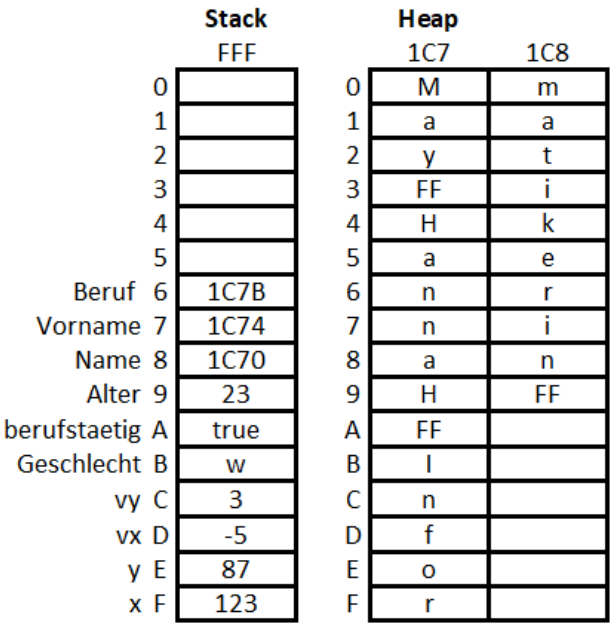


Abbildung 1.2.: einfache Variablen und Zeichenketten (String)

In Delphi werden - hinter dem Schlüsselwort VAR - nicht nur die Variablen vereinbart, sondern jeweils ein Typ festgelegt, der bei der Programmausführung nicht mehr geändert werden kann. Dabei steht INTEGER für ganze Zahlen, CHAR für einen Buchstaben, BOOLEAN für einen Wahrheitswert. Man sieht in der Grafik, dass diese einfachen Datentypen im Stack abgelegt werden. Die linke Spalte soll das „Wörterbuch“ andeuten.

1.3. Strukturierte Datentypen

Die **Zeichenketten (STRING)** gehören bereits zu den strukturierten Datentypen und werden als Referenztypen verwaltet. Das bedeutet, im Stack steht nicht der jeweilige Wert, sondern nur eine Referenz auf den Platz im Heap, wo der Wert gespeichert wird. Eine Referenz ist nichts anderes als eine Speicheradresse. Für die Verwendung dieser Speicheradresse gibt es unterschiedliche Methoden. Man kann eine feste Länge für alle Zeichenketten festlegen, etwa durch **STRING[20]**. Dann sind alle Zeichenketten im Speicher 20 Buchstaben lang. Das bedeutet, dass kürzere Worte genauso viel Speicherplatz einnehmen wie längere und das von Zeichenketten, die länger sind als 20 Buchstaben, nur die ersten 20 Buchstaben gespeichert werden. Der Speicher wird damit nicht optimal ausgenutzt. Deshalb gibt es einen andere Methode: Am Ende jeder Zeichenkette befindet sich ein Terminierungszeichen, das das Ende angibt. Häufig wurde dafür die Kombination FF (256 als Hexadezimalzahl, acht Einsen als Dualzahl) verwendet, andere Programmiersprachen wie C/C++ verwenden das Null-Byte, also 8 Nullen.

Ein weiterer strukturierter Datentyp ist die **Reihung (ARRAY)**. Die Reihung gehört zu den **statischen Datentypen**. Eine Reihung hat also immer **eine feste Länge**. Das Hinzufügen weiterer Elemente ist nur bis zur festgelegten maximalen Länge der Reihung möglich. Wird die festgelegte Höchstzahl unterschritten, enthält die Abbildung der Reihung im Speicher leere Elemente. Löscht man ein Element einer Reihung, dann müssen die darauf folgenden Element um einen Platz „vorgerückt“ werden. Bevor man ein Element einfügen kann, müssen, beginnend bei dem letzten nichtleeren Element der Reihung, alle Elemente hinter der Einfügeposition um ein Element nach hinten verschoben werden. ARRAYs wurden entwickelt, um Elemente gleichen Typs platzsparend abzulegen. Dabei haben alle Elemente die gleiche Länge. So definiert die folgende Variablenvereinbarung eine Reihung von sechs Namen, die jeweils die Länge 7 haben: **VAR namen: ARRAY[0..5] of STRING[7]**.

Man erkennt sehr schön die Probleme dieses

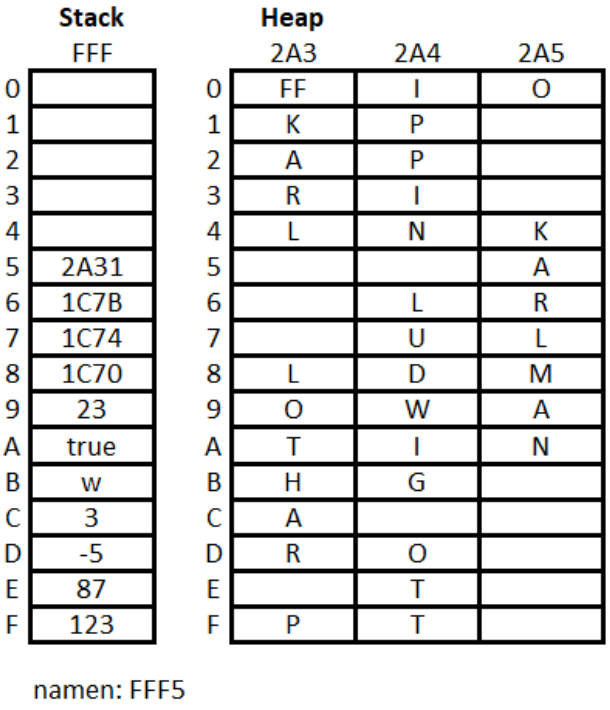


Abbildung 1.3.: Reihung (Array)

Datentyps. Da jedes Element gleich lang ist, führen kurze Namen wie Karl zu ungenutzten Lücken im Speicher. Andererseits passt ein Name wie Karlmann nicht in das feste Schema von sieben Buchstaben, der letzte Buchstabe wird abgeschnitten.

Unklar ist zunächst, warum bei sechs Elementen die Zählung des Index mit 0 beginnt und mit 5 aufhört. Diese Vereinbarung ermöglicht einen besonders einfachen Zugriff auf jedes einzelne Element der Reihung. Die Formel dafür lautet: Datensatzadresse = Startadresse + Index · Länge eines Datensatzes. In unserem Beispiel finden wir den ersten Datensatz mit Index 0 an der Adresse $2A31 + 0 \cdot 7 = 2A31$. Den fünften Datensatz (OTTO) mit Index 4 finden wir an der Adresse $2A31 + 4 \cdot 7 = 2A4D$.

Die für uns ungewohnte Zählung war in der Frühzeit der Computerentwicklung durchaus sinnvoll, weil sie einen einfachen Zugriff auf einzelne Elemente einer Reihung ermöglichte. Damals konnte der Programmierer noch selbst lesende und schreibende Zugriffe (PEEK und POKE) auf einzelne Speicherstellen durchführen. Solche Rechte gibt es heute noch teilweise bei Steuerungsanlagen, die Maschinen steuern, aber nicht mehr bei handelsüblichen Compu-

tern, die eine ausgeklügelte Speicherverwaltung verwenden. Programme mit direktem Speicherzugriff stehen im direkten Widerspruch zu der Möglichkeit, den Speicher eines PCs aufzurüsten. Also werden solche Zugriffe durch das Betriebssystem unterbunden. Das führt zu einem Widerspruch: Einerseits lassen viele Programmiersprachen Variablenvereinbarungen mit statischen Reihungen zu, andererseits wird der Speicher aber längst dynamisch verwaltet und es gibt Befehle wie SetLength in Delphi, die die Anzahl der Element einer Reihung während der Laufzeit eines Programms verändern.

Neben einer einfachen Reihung gibt es **zweidimensionale Reihungen** mit einer festen Anzahl von Spalten und Zeilen und einer festen Datensatzlänge. Auch hier beginnt jeweils die Zählung mit Null, d.h. die erste Zeile hat den Index Null und die erste Spalte hat ebenfalls den Index Null. Im Speicher wird ein Bereich von Zeilenzahl · Spaltenzahl · Datensatzlänge reserviert. In diesem Bereich werden zunächst die Elemente der ersten (0.) Zeile abgelegt, dann die Elemente der zweiten (1.) Zeile usw. Der Zugriff auf ein spezielles Element in Zeile n und Spalte m erfolgt mit der Formel Datensatzadresse = Startadresse+$(n-1)$·Spaltenzahl·Datensatzlänge+$(m-1)$·Datensatzlänge. Dazu ein Beispiel: Gegeben sei eine zweidimensionale Reihung VAR tabelle = ARRAY[0..4,0..6] of STRING[10] mit fünf Spalten und sieben Zeilen. Wir möchten auf das Element in der 3. Spalte und der 5. Zeile zugreifen. Das Element hat den Index [2,4]. Die Datensatzadresse erhält man mit der Formel Startadresse $+4 \cdot 5\cdot$ Datensatzlänge $+2\cdot$ Datensatzlänge. Das bedeutet, unser Element ist das 23. in der Reihenfolge der Element.

Neben der Reihung gibt es als strukturierten Datentyp den **Verbund** oder **RECORD**. Während die Reihung grundsätzlich Elemente gleichen Typs enthält, sind im Verbund auch Element unterschiedlichen Typs zulässig, also z.B. Zeichenketten, Zahlen und Wahrheitswerte. Auch der Verbund ist ein statischer Datentyp, d.h. bei der Vereinbarung wird der benötigte Speicherplatz festgelegt.

1.4. Dynamische Datentypen

Statische Datentypen sind nicht in der Lage, eine gute Auslastung des Arbeitsspeichers sicherzustellen, weil die Größe der Reihungen und Verbünde vor der Programmausführung festgelegt werden muss. Die Informatiker suchten deshalb nach einer Alternative und erfanden die dynamischen Datentypen.

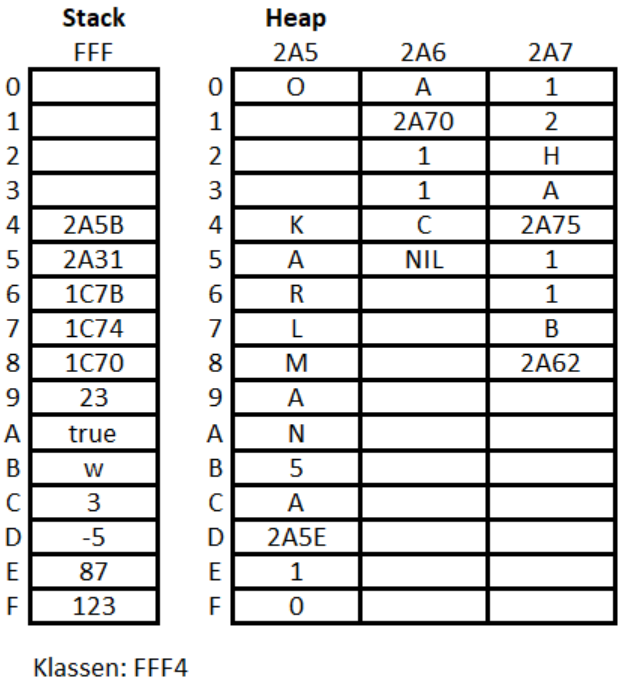


Abbildung 1.4.: Dynamische Liste

Dazu wurde die Idee der Referenz, also eines Zeigers auf eine Speicherzelle, gewissermaßen vervielfältigt. Zeichenketten und Reihungen werden im Stack als Referenz abgelegt, d.h. im Stack wird eingetragen, an welcher Speicherstelle die Zeichenkette und Reihung beginnt. Bei dynamischen Datentypen wird dieses Prinzip auf die Verbindung zwischen den einzelnen Datensätzen übertragen: Am Ende jedes Elements steht eine Referenz, die angibt, wo das nächste Element beginnt, bzw. die das letzte Element der Datenstruktur markiert. Damit kann der Speicherplatz dynamisch an den jeweiligen Bedarf angepasst werden. Das erklärt die Bezeichnung „dynamische Datentypen“.

Eine **Liste** ist das klassische Beispiel für einen solchen dynamischen Datentyp. An dieser Stelle ist ein Hinweis notwendig. Wenn in diesem Abschnitt von einer Liste die Rede ist,

sprechen wir von einer einfach verketteten Liste. Das ist nicht dasselbe wie eine Liste in Snap!, die wesentlich weiter entwickelt ist. Die Größe der einzelnen Elemente unserer verketteten Liste kann beliebig gewählt werden. Eine Liste erlaubt das Einfügen von Elementen, ohne dass wie bei der Reihung alle Elemente dahinter verschoben werden müssen. Die klassische Liste verfügt über eine Referenz auf dem Stack, auch Anker genannt. Diese Referenz zeigt auf das erste Element der Liste. Am Ende des ersten Elements steht eine weitere Referenz (bei Listen meist Zeiger genannt) auf das nächste Element usw. Die Element können im Heap hintereinander stehen, können aber auch in irgend einem freien Speicherbereich platziert werden. Ein direkter Zugriff auf eines der hinteren Listenelemente ist nicht möglich, man man jedes Mal die Liste von ihren Anker aus durchlaufen. Um die Länge einer Liste zu bestimmen, muss man beim Durchlaufen die Elemente Zählen. In den klassischen Programmiersprachen gelten Listen als besonders anspruchsvoll. So behandelt das Delphi 7 Kompendium bei einem Umfang von ca. 1200 Seiten Listen nur auf 18 Seiten, beginnend mit Seite 721. Es finden sich dort einige Ausführungen über Zeiger (Pointer) sowie einfach und doppelt verlinkte Liste.

Snap! dagegen behandelt die Listen als grundlegenden strukturierten Datentyp (neben der Zeichenkette). Die Snap!-Listen sind nicht nur dynamisch in der Größe an den jeweiligen Bedarf anpassbar, sondern sind zusätzlich indiziert, d.h. es lässt sich auf jedes Listenelement direkt über dessen Index zugreifen. Da die Berechnungen für die Datensatzadresse entfallen, beginnt die Zählung von Snap!-Listen auch nicht mit Null, sondern mit Eins. Allerdings sind auch die Snap!-Listen Referenztypen. Das hat Konsequenzen für das Kopieren von Listen. Ein einfaches

```
set MyList to list {2; 4; 9}
set My2ndList to MyList
```

erzeugt keine echte Kopie, weil nicht die Liste kopiert wird, sondern nur die Referenz. Anschließend sind die beiden Listen identisch, d.h. Veränderungen der einen Liste wirken sich sofort auf die andere aus.

Weitere dynamische Datentypen sind **Stapel** und **Schlange** (dazu in mehr in Kapitel 5) und der **Binärbaum** (in Kapitel 6). Theoretisch könnte man Stapel, Schlange oder Binärbaum auch mit Hilfe einer statischen Reihung realisieren, aber dazu müsste man den maximal benötigten Speicherplatz im Vorhinein reservieren.

Im Abitur steht eine weitere dynamische Datenstruktur in Form der Klasse **Dynamische Reihung** zur Verfügung. Man findet die Beschreibung der Dynamischen Reihung in den Hinweisen für die schriftliche Abiturprüfung im Fach Informatik in Niedersachsen. Sie wurde 2015 eingeführt, um einen Nachteilsausgleich für die JAVA-Programmierer zu schaffen, die nicht über ein so mächtiges Werkzeug wie unsere Listen in Snap! verfügen. Die Nummerierung der Elemente beginnt mit dem Index 0, wie bei den statischen Reihungen.

Die mögliche Implementierung in Snap! muss sauber von den Datentypen und ihren Eigenschaften getrennt werden. Statische Reihung, Verbund, einfach verkettete Liste, Stapel, Schlange, Binärbaum können in Snap! nur mit Hilfe von indizierten Snap!-Listen implementiert werden. Das bedeutete aber nicht, dass diese Datentypen die gleichen Eigenschaften wie Snap!-Listen hätten.

1.5. Typisierung

Klassische objektorientierte Programmiersprachen wie Delphi oder Java unterscheiden strikt zwischen verschiedenen Datentypen. Wir haben in den Beispielen in den vorigen Abschnitten gesehen, dass bei jeder Vereinbarung von Variablen der zugehörige Datentyp angegeben werden muss. Das hängt damit zusammen, dass diese Sprachen 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 deshalb ein Delphi-Programmierer angeben, welcher Datentyp jeweils vorliegt, also

ob ein STRING, eine INTEGER- oder eine REAL-Zahl vorliegt usw. Will er eine Zahl z.B. auf dem Bildschirm ausgeben, so muss er sie zunächst in eine Zeichenkette umwandeln. Will er mit einer Eingabe rechnen, so muss er zunächst die Zeichenkette in eine Zahl umwandeln. zu den häufigsten Delphi-Befehlen gehören deshalb IntToStr() zum Umwandeln einer Ganzzahl in eine Zeichenkette und StrToInt() zum Umwandeln einer Zeichenkette in eine Ganzzahl.

Snap! hat kein so starres Datenkonzept. Die Zahlen werden ziffernweise dargestellt, so dass man sowohl mit ihnen rechnen als auch z.B. mit `letter i of text` auf einzelne Ziffern zugreifen kann. Diese Flexibilität erkauft man mit einer höheren Verantwortung des Programmierers, der sich nicht mehr darauf verlassen kann, dass das Programm falsche Typzuweisungen mit einer Fehlermeldung quittiert. In 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.

2. Rekursion und Grafik

In diesem Kapitel lernen Sie

- was die Eigenschaften von Rekursion sind,
- wie man eine rekursive Operation implementiert,
- wie man eine Reihe von Messwerten graphisch darstellt.

2.1. Rekursion

Unter Rekursion[1] 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:

Abbildung 2.1.: grundsätzlicher Aufbau einer Rekursion

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 weiter delegiert.

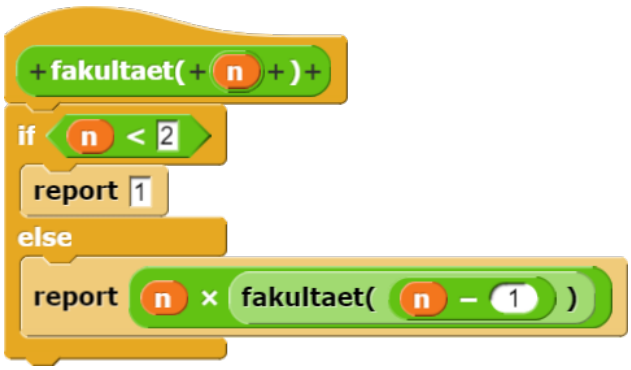


Abbildung 2.2.: Rekursion Fakultät

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. Im Englischen spricht man von **base case**. 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.

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

[1] Eine erste Fassung dieses Abschnitts stammt von Niklas Euler

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!:

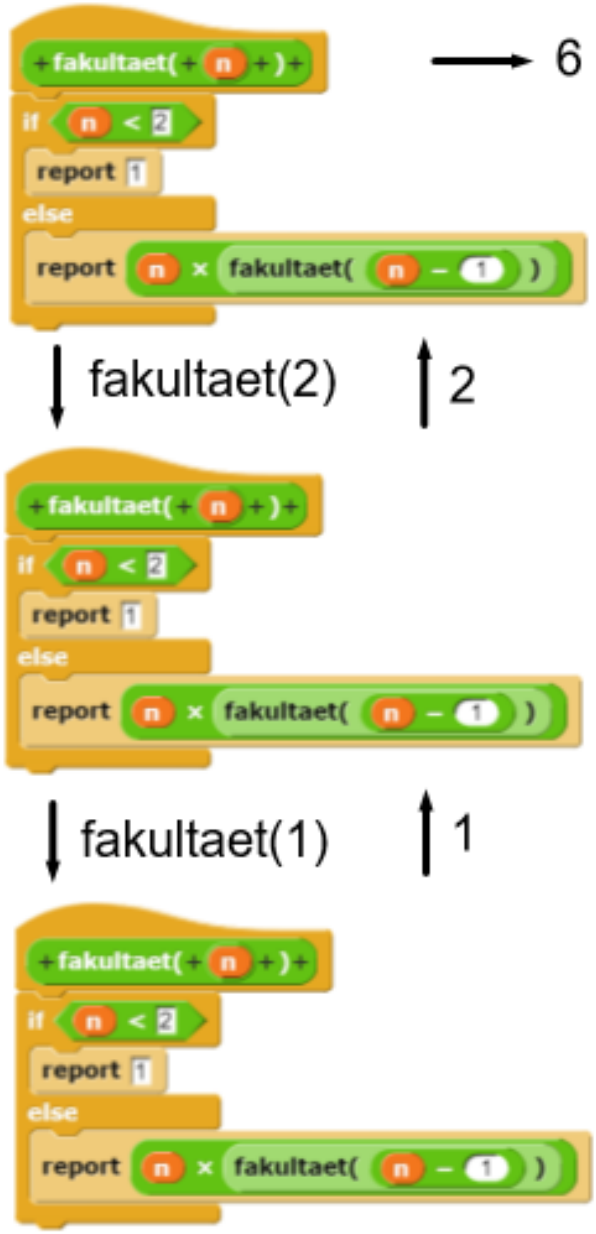


Abbildung 2.3.: Prinzip der Rekursion

Rufen wir den Fakultäts-Block mit dem Parameter 3 auf, so stellt der Block zunächst fest, dass die Abbruchbedingung noch nicht erreicht ist. Zur Berechnung von 3! benötigt der Block aber 2!. Er startet eine neue Instanz mit dem Parameter 2. Die neue Instanz stellt ebenfalls fest, dass die Abbruchbedingung noch nicht erreicht ist, dass sie aber zur Berechnung von 2! den Wert von 1! benötigt. Sie startet die dritte Instanz mit dem Parameter 1. Die dritte Instanz stellt fest, dass die Abbruchbedingung erreicht ist, und meldet das Ergebnis 1 an die zweite Instanz. Mit diesem Ergebnis kann die zweite Instanz ihr Ergebnis 2 berechnen und meldet es an die erste Instanz. Diese kann nun 3!=6 berechnen und ausgeben.

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.

Grundsätzlich besteht die Idee der Rekursion in der Zerlegung des Problems in verschiedene Ebenen, wobei immer nur die aktuelle Ebene behandelt wird und die übrigen Ebenen durch den Selbstaufruf einer Funktion mit geänderten Parametern (z.B. n+1, n-1, leftTree, rightTree, usw.) berücksichtigt werden, wobei die Anzahl solcher Aufrufe durch eine Abbruchbedingung begrenzt werden muss.

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).

Das Gegenteil von rekursiv wäre iterativ. Bei der **iterativen Programmierung** verwenden

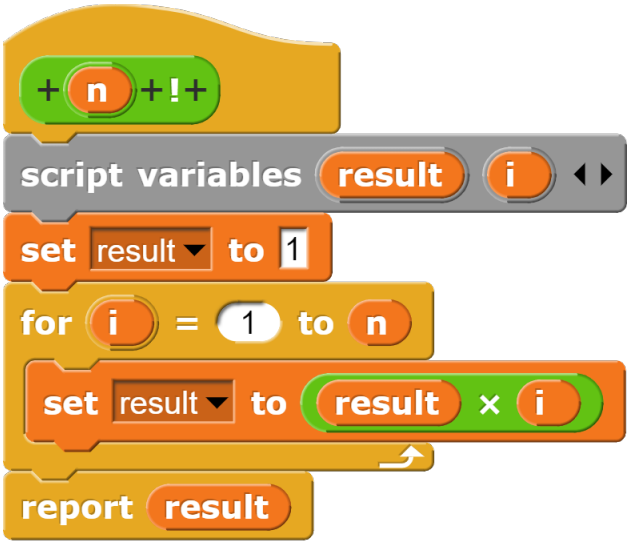


Abbildung 2.4.: Fakultät iterativ

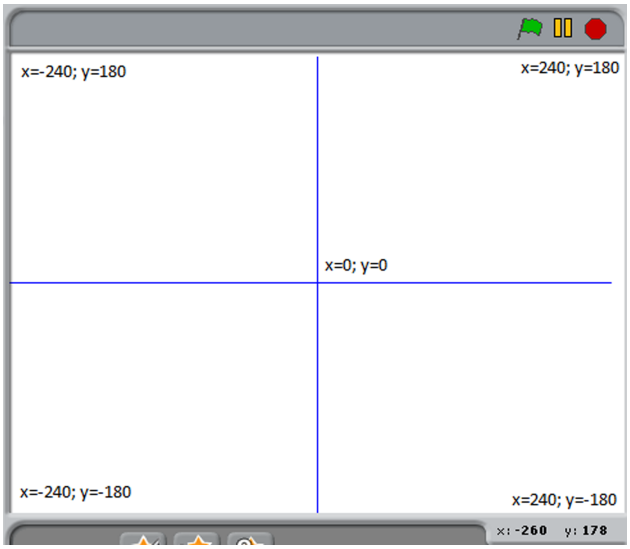


Abbildung 2.5.: Bildschirmkoordinaten

wir Schleifen, um Anweisungen oder Sequenzen (Anweisungsfolgen) zu wiederholen und so ein Ergebnis zu berechnen.

Die Fakultät einer Zahl kann auch mit Hilfe einer FOR-Schleife berechnet werden, indem nacheinander die Zahlen von 1 bis n miteinander multipliziert werden.

2.2. 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.

Snap! 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 $x_m$min $= y_m$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 Xm2Xb zur Umrechnung der mathematischen x-Koordinaten in die Bildschirm-x-Koordinaten und einen Block Ym2Yb 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:

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

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

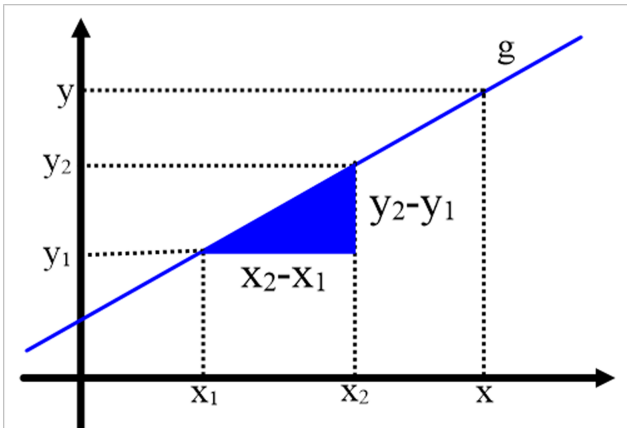


Abbildung 2.6.: Geradensteigung

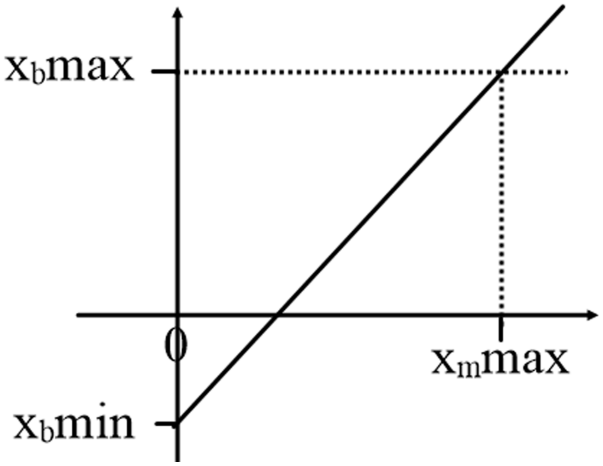


Abbildung 2.7.: mathematische in Bildschirmkoordinaten

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 Xm2Xb:

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

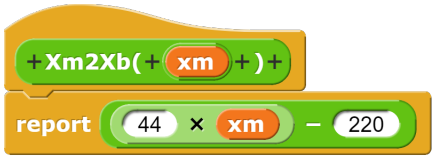


Abbildung 2.8.: mathematische x- in Bildschirmkoordinaten

Entsprechend berechnet man die Funktion zum Umrechnen der y-Koordinaten. Allerdings wollen wir mehrere Messreihen mit je 10 Elementen zeichnen und den Zeichenplatz voll ausnutzen. Dazu verwenden wir eine globale Variable **YmMax** für den größten Messwert in unseren Listen. Dann haben wir folgende Formel zur Umrechnung der y-Koordinaten:

$$y - y_bmin = \frac{y_bmax - y_bmin}{YmMax - 0} \cdot (x - 0)$$

$$y - (-160) = \frac{160 - (-160)}{YmMax - 0} \cdot (x - 0)$$

aufgelöst nach y ergibt sich für die Funktion Xm2Xb:

$$y = \frac{320}{YmMax} \cdot x - 160$$

Wir müssen darauf achten, dass wir der globale Variable **YmMax** den richtigen Wert zuweisen, bevor wir zeichnen, sonst erhalten wir nur eine waagrechte Linie bei y=-160.

Abbildung 2.9.: mathematische y- in Bildschirmkoordinaten

Der eigentliche Zeichenblock ist dann recht einfach: Wir gehen zu Ursprung der Bildschirmkoordinaten, setzen den Stift (Pen) ab und gehen dann mit einer FOR-Schleife die Liste mit den Messwerten durch. Dabei laufen die x-Koordinaten von 1 bis 10 und die y-Koordinaten durch die entsprechenden Messwerte. Abschließend wird der Stift wieder angehoben. Bei der Darstellung mehrerer

Messreihen kann man mit einem zweiten Parameter auch die Farbe einstellen, in der gezeichnet werden soll.

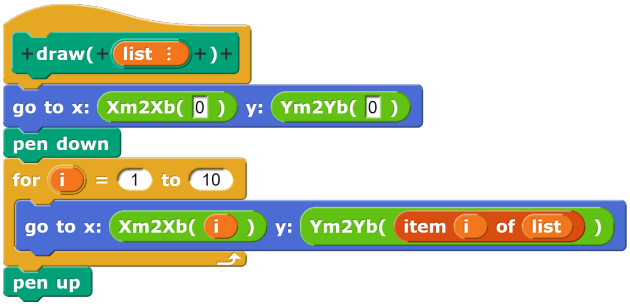


Abbildung 2.10.: Zeichnen von Messwerten

Für die Zeitmessung können wir zwei Blöcke aus der Sensing-Palette nutzen: **reset timer** startet die interne Uhr neu und nach dem Ende des Experiments können wir mit dem Reporter **timer** die verstrichene Zeit abfragen.

2.3. Kreis-Kostüm

Kostüme werden durch Snap! in einer spezifischen Form gespeichert: Sie haben einen Namen (`name`), eine Breite (`width`), eine Höhe (`height`) und eine Liste von Pixeln (`pixels`). Die Pixelliste ist keine Tabelle, sondern eine einfache Liste, die erst mit Hilfe der Breite in eine Tabelle umgesetzt wird. Ein Pixel besteht aus einem Rot-Wert, einem Grün-Wert, einem Blau-Wert und einem Wert für den Deckungsgrad. Alle diese Werte liegen im Bereich von 0 bis 255.

Abbildung 2.11.: New Costume

Unter Sensing findet sich ein Block zum Auslesen dieser Eigenschaften, und es gibt mit `new costume` einen Block, der aus einer Pixelliste eine Sprite erstellen kann, wenn wir Breite und Höhe als Parameter vorgeben.

Abbildung 2.12.: Abstand zweier Punkte

Ein Kreis ist mathematisch definiert als die Menge der Punkte, die einen bestimmten Abstand vom Mittelpunkt haben. Dieser Abstand ist der Radius. Daraus ergibt sich: die Kreisfläche ist die Menge aller Punkte, deren Abstand vom Mittelpunkt kleiner ist als der Radius. Diesen Abstand können wir mit dem Satz des Pythagoras berechnen: $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.

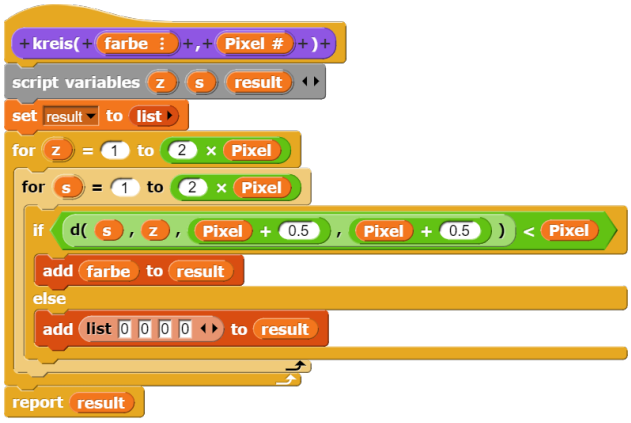


Abbildung 2.13.: Pixelliste für einen Kreis

Der Block `kreis` hat als ersten Parameter die Farbe als Liste mit dem Rot-, Grün- und Blau-Wert und dem Wert für die Undurchsichtigkeit als Elementen. Der zweite Parameter ist der Radius in Pixeln. Der Block erzeugt die Pixel für ein Quadrat mit einer Seitenlänge, die doppelt so groß ist wie der Radius. Daraus ergeben sich die Koordinaten für den Mittelpunkt $(2 \cdot Pixel + 0,5 | 2 \cdot Pixel + 0,5)$. Alle Pixel, die dichter am Mittelpunkt sind als der Radius, werden mit der Farbe belegt, alle anderen mit (0|0|0|0), also „weiß durchsichtig“.

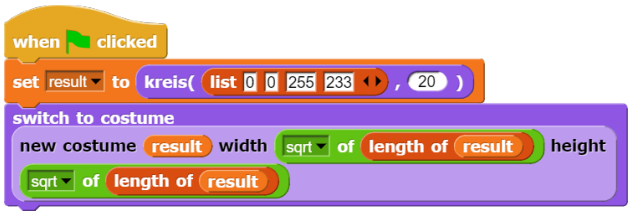


Abbildung 2.14.: Erstellen des Kreis-Kostüms

Da es sich um eine quadratische Bitanzahl handelt, lassen sich Breite und Höhe jeweils als Wurzel aus der Anzahl der Listenelemente bestimmen.

2.4. Aufgaben

Aufgabe 2.1 *Gegeben ist das Struktogramm einer Operation* `aussortieren` *mit dem Parameter s, wobei s eine Liste ist. Erläutern*

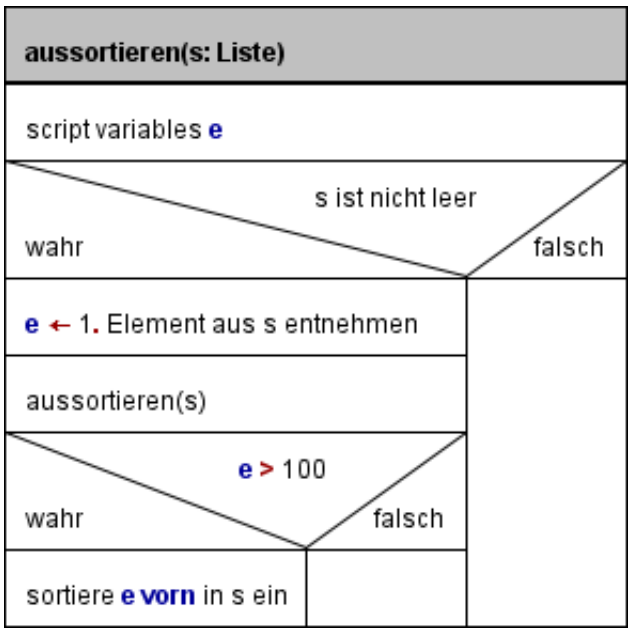


Abbildung 2.15.: Aussortieren

Sie die Funktionsweise der Operation `aussortieren(s: Liste)`.

Implementieren Sie die Operation `aussortieren(s: Liste)` *in der durch das Struktogramm gegebenen Fassung.*

Implementieren Sie einen Reporter `aussortieren(s: Liste)` *iterativ.*

Aufgabe 2.2 *Implementieren Sie eine Operation zur Berechnung des größten gemeinsamen Teilers (ggT) zweier Zahlen. Gehen Sie rekursiv nach den folgenden Aussagen vor. Der ggT von zwei positiven ganzen Zahlen a und b, a>b, ist:*

- *b, wenn a mod b gleich Null.*
- *ggT(b, a mod b) im anderen Fall.*

Aufgabe 2.3 *Ein Kamel soll optimal beladen werden. Das Kamel kann maximal 270 kg tragen. Aktuell sind Waren mit den folgenden Gewichten zu transportieren: 5, 18, 32, 34, 45, 57, 63, 69, 94, 98 und 121 kg. Nicht alle Gewichte müssen verwendet werden; die 270 kg sollen aber möglichst gut, wenn nicht sogar ganz ohne Rest beladen werden. Die Operation* `beladeOptimal(kapazitaet: Ganzahl, vorrat: Liste): Liste` *erhält als Parameter die maximal tragbare Last (*`kapazitaet`*) und eine Liste von den aufzuteilenden Waren bzw. deren Gewichten (*`vorrat`*). Das Resultat ist eine Liste mit der Auswahl aus dem Vorrat, die der maximalen Belastbarkeit möglichst nahe kommt. Gehen Sie wie folgt rekursiv vor: Für jedes vorhandene Gewicht g aus dem Vorrat soll das Problem vereinfacht werden. Dazu wird dieses Gewicht probehalber aufgeladen:*

`set tmpLadung to beladeOptimal (kapazitaet − g, vorrat ohne g)`

Danach wird das beste Resultat `tmpLadung + g` *gesucht, das kleiner ist als die Kapazität, und als Liste zurückgegeben. Behandeln Sie in der Operation beladeOptimal() zunächst die Abbruchbedingungen:*

- *Vorrat leer*
- *alle vorhandenen Gewichte sind zu schwer*

Tipp: Schalten Sie zum Testen der Operation den Turbo-Modus ein! Das Einschalten können Sie automatisieren, indem Sie nach der grünen Flagge aus der SENSING-Palette den `set video capture to`*-Block einfügen. Der Block kann nicht nur die Videoaufzeichnung starten, sondern er verbirgt hinter dem schwarzen Pfeil nach unten weitere Systemoptionen, z.B.* `turbo mode` *und* `flat line ends`*. Einfach auf* `turbo mode` *einstellen und ein* `true` *in das leere Sechseck ziehen, schon läuft Snap! schneller.*

3. Listen

In diesem Kapitel lernen Sie

- was First-Class-Datenstrukturen sind
- welche Blöcke zur Bearbeitung von Listen Snap! zur Verfügung stellt
- wie man mit Funktionen höherer Ordnung rekursive Sortierverfahren elegant implementiert

3.1. Erstellen von Listen

Listen sind der zentrale Datentyp in Snap!. Sie sind dynamische Datenstrukturen wie Listen in anderen Programmiersprachen, sie erlauben den gezielten Zugriff auf einzelne Elemente wie ARRAYs, sie können unterschiedliche Datentypen enthalten wie RECORDs. Sie können auch weitere Liste enthalten und

Eine Reihung (ARRAY) ist ein Datentyp, für den ein fester Bereich im Speicher reserviert und in gleich große Teilstücke aufgeteilt wird(siehe Kapitel 1.2). Snap! kennt den Datentyp Reihung nicht, wir verwenden statt dessen dynamische Listen, deren Größe automatisch an den Inhalt angepasst wird.

Abbildung 3.1.: List-Block

Der Reporter `list` liefert eine „anonyme“ Liste, d.h. eine Liste ohne ihr gleichzeitig einen Namen zu geben. 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.

In Snap! sind Listen **First-Class-Datenstrukturen**. Eine Datenstruktur ist von erster Klasse, falls sie die folgenden Bedingungen erfüllt:

- sie kann in Programmvariablen gespeichert werden,
- sie kann als Parameter an Blöcke übergeben werden,
- sie kann als Rückgabewert von Blöcken dienen,
- sie kann zur Laufzeit eines Programms erstellt werden und
- sie hat eine eigene Identität (unabhängig vom Namen)

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

Eine der häufigsten Operationen mit Liste ist das Sortieren. Eine Reihe von Sortierverfahren, die zu den Grundrezepten gehören, sind im Band für die 11. Klasse beschrieben (SelectionSort und InsertionSort, siehe auch im Anhang zu diesem Band).

3.2. Command-Blöcke für Listen

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`.

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, der größer ist als die Länge der Liste, bleibt der Block wirkungslos.

Abbildung 3.2.: Command-Blöcke für Listen

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 größer ist als die Länge der Liste, wird das neue Element hinten angefügt.

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. **Achtung:** Wird eine Zahl als Index angegeben, die größer ist als die Länge der Liste, werden leere Element eingefügt, bis der angegebene Index erreicht ist, und das letzte Element enthält den Inhalt.

3.3. Reporter-Blöcke für Listen

`Numbers from start to ende` liefert eine Liste mit den ganzen Zahlen von Start bis Ende. Je nach Einstellung wird aufwärts oder abwärts gezählt.

`Something in front of list` erzeugt eine neue Liste mit dem ersten Parameter als erstem Element.

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.

`All but first of list` liefert eine neue Liste, die alle Elemente der Ausgangsliste außer dem ersten enthält. Die ursprüngliche Liste wird nicht verändert.

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 wort` 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.

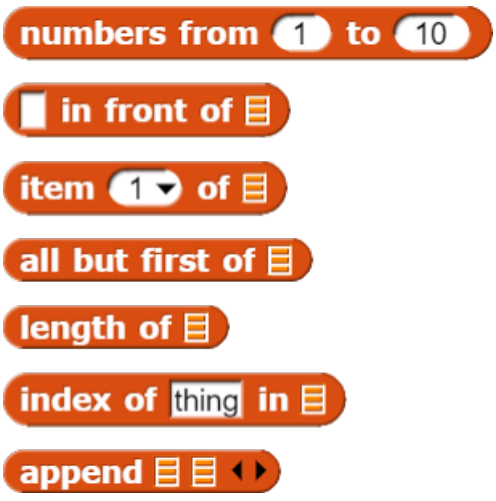


Abbildung 3.3.: Reporter-Blöcke für Listen

`Index of thing in list` liefert den Index (die Platznummer) des ersten Vorkommens des angegebenen Elements in der Liste. Dieser Befehl ist besonders nützlich bei der Ersetzung

von Elementen einer Liste aus Elementen anderer Liste mit gleichem Index (vgl. z.B. die verbesserte Version eines DEA oder die Huffman-Codierung).

`Append` führt Listen zu einer neuen Liste zusammen. Über die schwarzen Pfeile kann die Anzahl der Listen eingestellt werden.

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.

3.4. Predicate

Abbildung 3.4.: Predicate und Schleifen für Listen

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

Allerdings muss man hier aufpassen, wenn man in der Liste Zeichenketten aus Ziffern speichert. Das Prädikat unterscheidet nicht zwischen `0` und `00`. Es sieht nur Ziffern, folgert, dass es sich um Zahlen handelt, und setzt 0=00. Führende Nullen werden nicht berücksichtigt. Eine mögliche Abhilfe wird im Abschnitt über die Huffman-Decodierung erläutert.

Abbildung 3.5.: Contains mit Zahlen

Neu ist der Prädikatsblock `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`.

Der Gleichheitsoperator = überprüft seit der Version 6.0 bei Listen nicht mehr nur die Referenz, sondern auch die Gleichheit von Listen.

`For each item in List` funktioniert ähnlich wie eine `FOR`-Schleife, nur dass keine Zählvariable verwendet wird.

3.5. Funktionen höherer Ordnung

Abbildung 3.6.: Funktionen höherer Ordnung

Als Funktionen höherer Ordnung bezeichnet man Blöcke, die andere Blöcke als Parameter verwenden. Die vier Funktionen map, keep, find und combine haben jeweils zwei Parameter, eine Liste und ein Predicate. Das Predicate hat einen grauen Ring. Die Auswertung eines beliebigen Predicate-Blocks liefert wahr oder falsch. Ist der Block in einen grauen Ring eingeschlossen, so liefert die Auswertung den

Block selbst. Diese Eigenschaft erlaubt den Funktionen höherer Ordnung, den Block mehrfach auszuwerten, für jedes Element der Liste. In die freien Stellen werden die Elemente der Liste eingesetzt. Sofern wir nicht durch value, index, list genauer angeben, was jeweils eingesetzt werden soll, können wir die Felder einfach frei lassen. **Diese freien Felder kennzeichnen wir durch eckige Klammern** (im Unterschied zu Listen, für die wir geschweifte Klammern verwenden, und logische Einheiten, für die wir runde Klammern verwenden).

Der `Keep`-Block dient zur Auswahl von Elementen. So liefert

`keep item ringify([ ]  >3) from list {4; 2; 1; 5 }`

die Liste {4; 5}. Dabei bleibt der erste Parameter des Größer-Blocks leer. Wir kennzeichnen das durch die eckigen Klammern. Hier werden nacheinander die Listenelement eingesetzt. In die neue Liste werden nur diejenigen übernommen, die die Bedingung erfüllen.

Der `Map`-Block führt eine Operation mit allen Elementen der Liste aus und liefert eine neue Liste mit den Ergebnissen.

`map ringify([ ]) over myList`

liefert zwar eine echte Kopie von myList, nicht nur eine Kopie der Referenz wie beim SET-Befehl. Ab Version 8.0 sollte man sich aber angewöhnen, für das Kopieren von Listen den `id`-Befehl zu verwenden, weil der auch von geschachtelten Listen echte Kopien liefert.

Der `Combine`-Befehl kombiniert schließlich alle Elemente einer Liste. Als Parameter können der Plus-, der Mal-, der Join-, der AND- und der OR-Block eingesetzt werden.

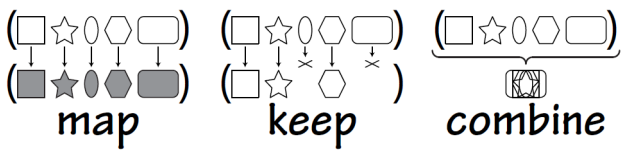


Abbildung 3.7.: Funktionen höherer Ordnung

Die Blöcke höherer Ordnung verfügen über einen kleinen Rechtspfeil, über den drei mögliche Eingabeparameter erreichbar sind: value steht für den Werte eines Listenelements, index für seinen Platz in der Liste und list für die Liste. Alle drei Parameter können bei der Formulierung von Bedingungen verwendet werden.

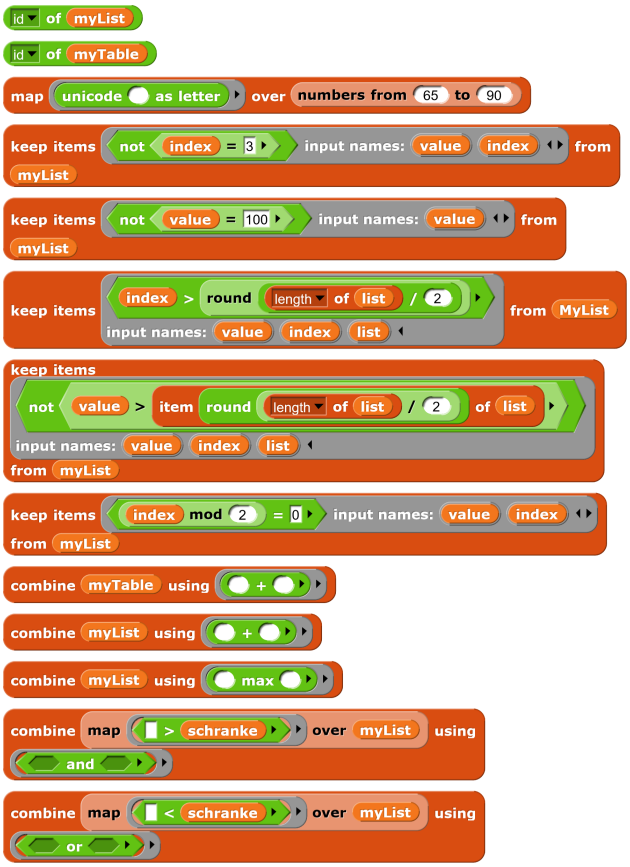


Abbildung 3.8.: Beispiele für Funktionen höherer Ordnung

Einige Beispiele für Funktionen höherer Ordnung:

`map ringify([ ]) over myList` liefert eine echte Kopie einer eindimensionalen Liste `myList`. **Achtung:** Dieser Befehl funktioniert zwar noch ab Version 8.0, aber man sollte sich angewöhnen, statt dessen den `id`-Block aus der Operatoren-Palette zu verwenden (er versteckt sich unter dem `sqrt of 10`-Block ganz unten). Im Unterschied zum `map`-Befehl liefert `id` eine sogenannte *deep copy*, d.h. eine echte Kopie einer verketteten Liste beliebiger Tiefe, also z.B. auch eines Baumes.

`map ringify (unicode [ ] as letter) over (numbers from 65 to 90)` liefert das Alphabet in Großbuchstaben.

`map ringify (join (b,[ ])) over codeliste` liefert die Codeliste, wobei vor jedes Element von Codeliste ein kleines `b` gesetzt wird.

`map ringify (map ringify (round`

`0) over numbers from 1 to 26) over numbers from 1 to 6` liefert eine Tabelle von 26 Spalten und 6 Zeilen, die mit Nullen initialisiert ist.

`keep items ringify (not (index=3)) from myList` liefert eine Liste ohne das 3. Element von `myList`.

`keep items ringify (not (value=100)) from myList` liefert eine Kopie von `myList` ohne das Element mit dem Wert 100.

`keep items ringify (index > round(length of list/2)) from myList` liefert die hintere Hälfte von `myList`.

`keep items ringify (not (index > round(lenght of list/2))) from myList` liefert die vordere Hälfte von `myList`.

`keep items ringify (value > item( round(lenght of list/2))) from myList` liefert eine Liste mit allen Elementen, die größer sind als das mittlere Element von `myList`.

`keep ringify (Eingabealphabet contains [ ]) from (split Eingabe by letter)` wandelt die Zeichenkette `Eingabe` in eine Buchstabenliste um und entfernt alle die Zeichen, die nicht in der Liste `Eingabealphabet` enthalten sind. Ausgegeben wird eine Liste mit den zulässigen Eingabezeichen.

`keep items ringify (index mod 2 = 0) from myList` liefert eine Liste mit den Elementen an den geraden Plätzen von `myList`.

`combine myList using ringify ([ ] + [ ])` liefert die Summe von `myList`.

`combine (map ringify ([ ] > schranke) over myList) ringify ([ ] and [ ])` gibt an, ob alle Elemente von `myList` größer sind als `schranke`.

`combine (map ringify ([ ] < schranke) over myList) ringify ([ ] or [+])` gibt an, ob `myList` Elemente enthält, die kleiner sind als `schranke`.

`combine (numbers from 1 to n) using ringify ([ ] * [ ])` mit dem Parameter n liefert `n!`.

`combine tabelle using ([ ] + [ ])` liefert eine Liste mit den Zeilensummen einer zweidimensionalen Reihung namens Tabelle.

`combine (combine tabelle using ringify ([ ] + [ ])) using ringify ([ ] + [ ])` liefert die Summe aller Elemente einer zweidimensionalen Reihung namens Tabelle.

`combine myList using ringify ([ ]`*max*`[ ]))` liefert das größte Element einer Liste namens myList.

3.6. Aufgaben

Aufgabe 3.1 *Implementieren Sie eine Operation* ***`Summe`****, die die Summe einer Liste von Zahlen ausgibt.*

Aufgabe 3.2 *Implementieren Sie eine Operation* ***`Max`****, die das Maximum einer Liste von Zahlen ausgibt.*

Aufgabe 3.3 *Implementieren Sie eine Operation* ***`3Max`****, die die drei größten Zahlen einer Liste von Zahlen ausgibt.*

Aufgabe 3.4 *Implementieren Sie eine Operation* ***`entferneDoppel`****, die alle doppelten Einträge aus einer Liste entfernt.*

Aufgabe 3.5 *Implementieren Sie eine Operation* ***`lottoprognose`****, die einen Vorschlag für Lotto 6 aus 49 ausgibt.*

Aufgabe 3.6 *Gegeben ist eine Liste mit Zahlen, die aufsteigend sortiert sind. Ein „Plateau“ ist eine Folge gleicher Zahlen. Gesucht ist eine Operation* ***`laengstesPlateau`****, die die Länge des längsten Plateaus ausgibt. Beispiel: {5, 6, 6, 7, 10, 10, 10, 11, 15, 15, 20, 21, 25 }. Es gibt 3 Plateaus; das längste ist drei Felder lang.*

4. Sortierverfahren

In diesem Kapitel lernen Sie

- wie man iterative Sortierverfahren implementiert
- wie man zwei Listen im Reißverschlussverfahren zusammenführt („mergen“)
- wie man rekursive Sortierverfahren implementiert

4.1. 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):

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.

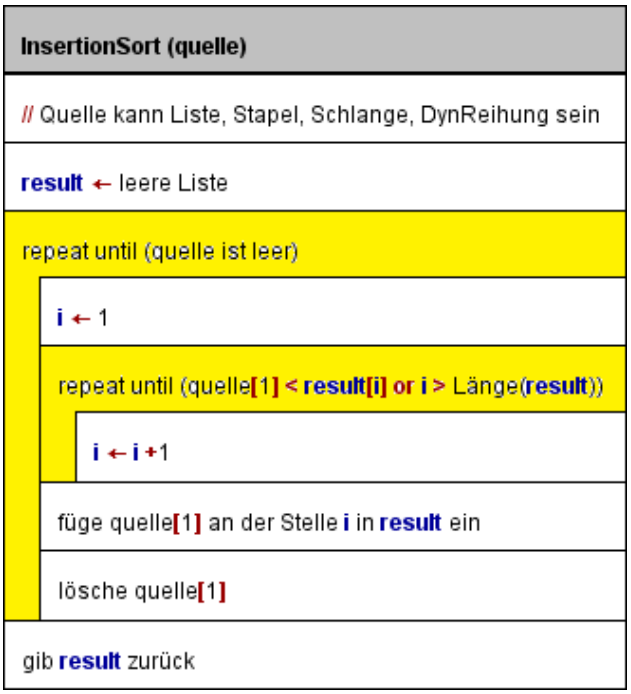


Abbildung 4.1.: InsertionSort

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:

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,

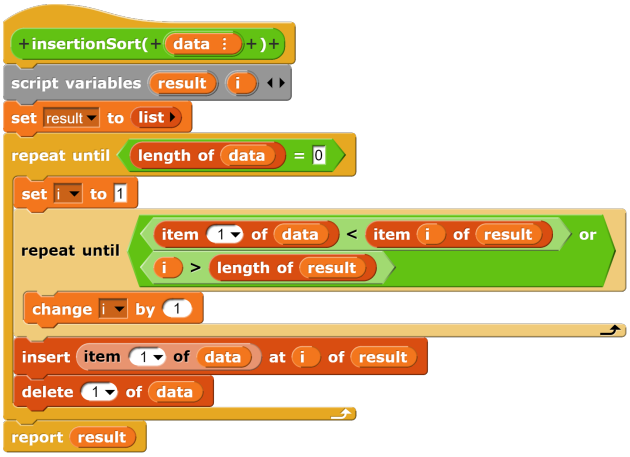


Abbildung 4.2.: InsertionSort

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.

Variation von InsertionSort: InsertionSort funktioniert nicht mit Listen, deren Elemente nicht nur Zahlen oder Zeichenketten, sondern wiederum Listen sind. Wenn die Elemente der Hauptliste nach einem bestimmten Element der Teillisten sortiert werden sollen, gibt es eine Fehlermeldung, weil `result` zu Beginn des Sortiervorgangs leer ist. Deshalb kann man nicht auf `item(i) of item(1) of result` zugreifen. Um Abhilfe zu schaffen, kann man vorab das erste Element aus `data` in `result` einfügen und in `data` löschen.

Ruft man den Block auf, so wird die als Parameter übergebene Liste gelöscht, weil die Übergabe als Parameter keine echte Kopie erzeugt, sondern nur eine Kopie der Referenz. Man kann dies verhindern, indem man nicht die zu sortierende Liste selbst, sondern eine Kopie übergibt, also `insertionSort(id of data)`.

4.2. 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 Sortiertverfahren seinen Namen.

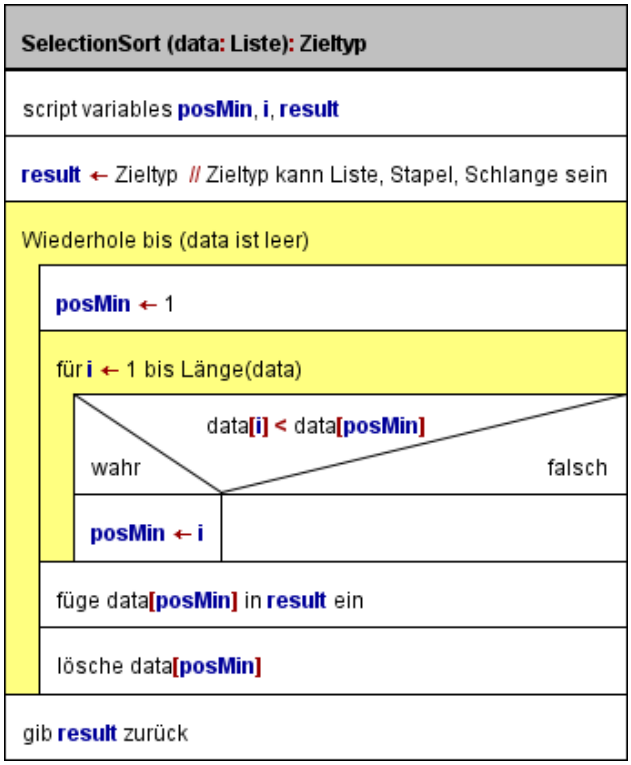


Abbildung 4.3.: Selektionsort

Wir benötigen drei Skriptvariable: `result` für das Ergebnis, eine Zählvariable `i` und ei-

ne 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.

Das Verfahren lässt sich einfach implementieren, wobei wir hier voraussetzen, dass das Ergebnis eine Liste ist.

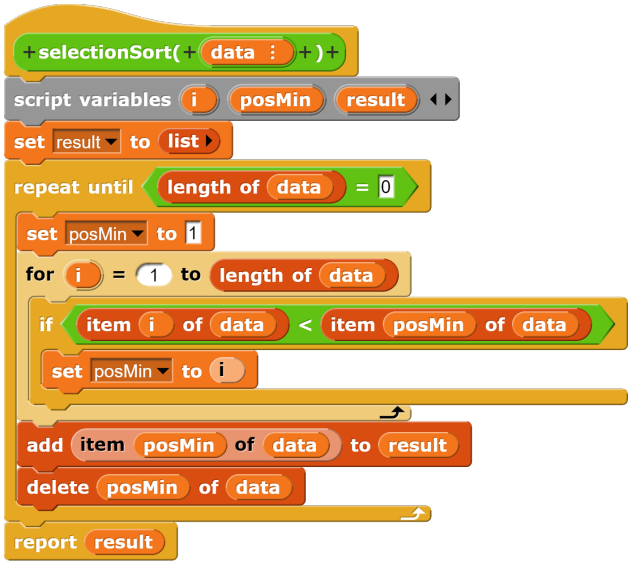


Abbildung 4.4.: Selektionsort

Ruft man den Block auf, so wird die als Parameter übergebene Liste gelöscht, weil die Übergabe als Parameter keine echte Kopie erzeugt, sondern nur eine Kopie der Referenz. Man kann dies verhindern, indem man nicht die zu sortierende Liste selbst, sondern eine Kopie übergibt, also `selectionSort(id of data)`.

4.3. 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 lediglich 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.

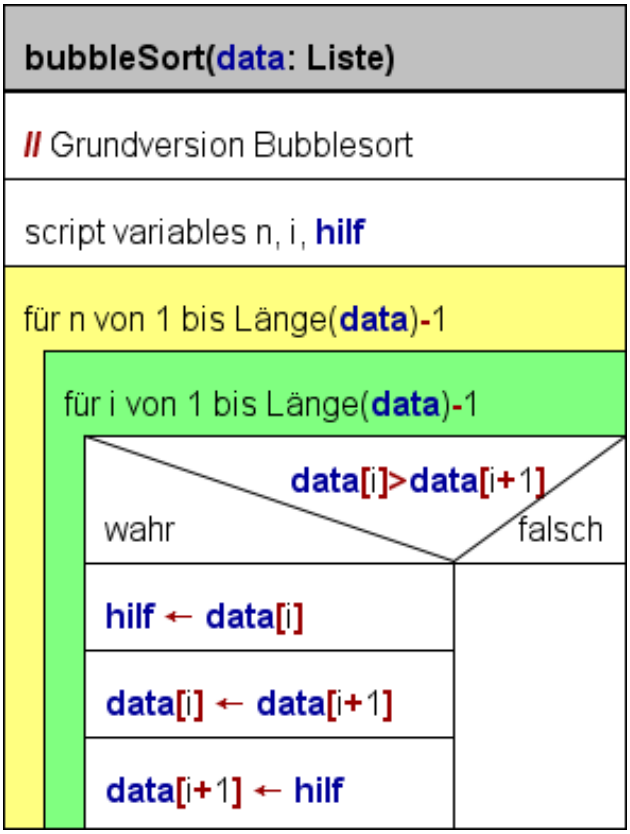


Abbildung 4.5.: BubbleSort-Grundform

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 getauscht ein, die in der äußeren Schleife abgefragt wird. Diese Variable wird zu Beginn jeden Durchgangs auf `false` gesetzt. Bei jedem Tauschvorgang wird sie auf `true` gesetzt. Die äußere Schleife wird so lange durchgeführt, bis `getauscht = false` ist.

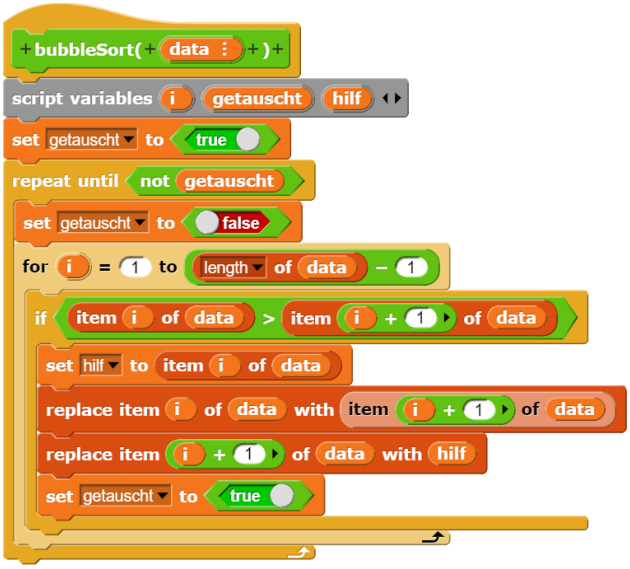


Abbildung 4.6.: optimiertes BubbleSort

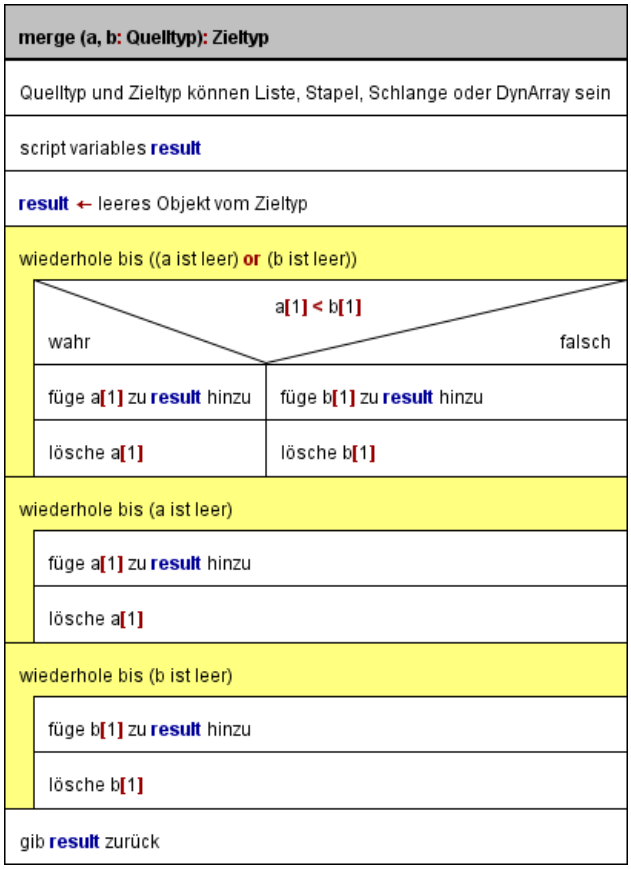


Abbildung 4.7.: Struktogramm Merge

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.

4.4. Grundrezept Merge

Als Mergen bezeichnet man das Zusammenfügen von zwei sortierten Datenstrukturen zu einer neuen, ebenfalls sortierten Datenstruktur. Dazu werden die beiden ersten Elemente verglichen. Das kleinere wird dann zum Ergebnis hinzugefügt und aus der ursprünglichen Struktur entfernt. Dieser Vorgang wird so lange wiederholt, bis eine der beiden Datenstrukturen leer ist. Anschließend wird der Rest in der andern Struktur angefügt. Wir trennen hier Merge und Mergesort, weil der Merge-Vorgang auch in anderen Aufgabenstellungen (auch in Abituraufgaben) vorkommt. Zulässige Datenstrukturen sind alle, die einen Zugriff auf das erste Element erlauben, also Listen, Dynamische Arrays, Stapel und Schlange. Damit ergibt sich folgendes Struktogramm:

Bei der Implementierung mit Listen können wir die „Restbearbeitung“ vereinfachen, indem wir den Append-Befehl verwenden. Eine der beiden Teillisten ist zu diesem Zeitpunkt bereits leer, was aber nicht weiter stört.

Variationen von Merge: Neben dem sortierten Mergen können z.B Warteschlangen auch abwechselnd zusammengefügt werden, um eine neue Warteschlange zu bilden. Es entfällt dann die IF-Verzweigung. Statt dessen werden aus beiden Quellstrukturen die ersten Elemente entnommen und gelöscht.

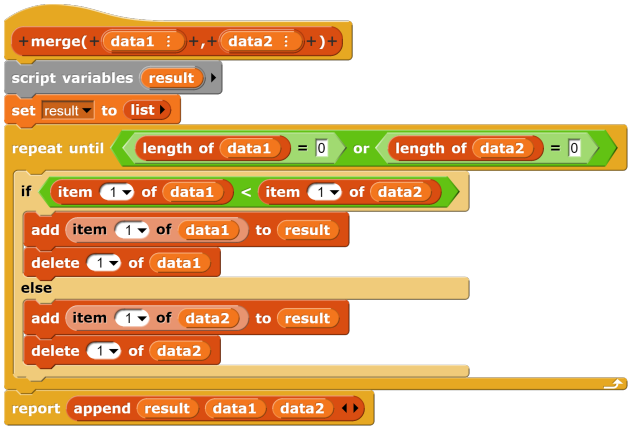


Abbildung 4.8.: Merge mit Listen

4.5. Grundrezept MergeSort

Mergesort ist ein rekursives Sortierverfahren. Eine Liste wird in kleinere Listen zerlegt, die einzeln sortiert und dann mit Hilfe des Merge-Algorithmus zu einer sortierten Liste zusammengesetzt werden. Als Abbruchbedingung verwendet man die Tatsache, dass Listen mit einem Element bereits sortiert sind.

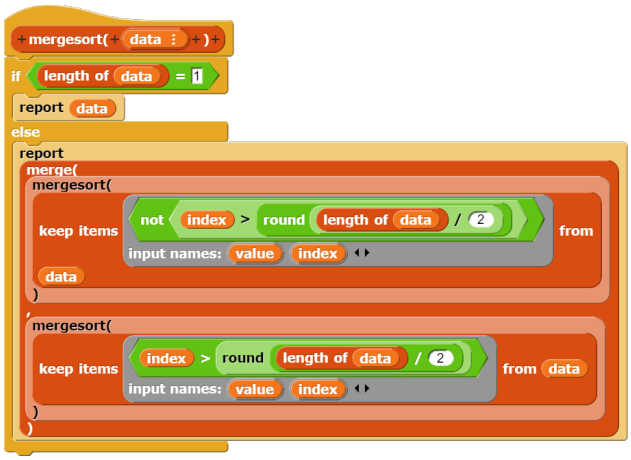


Abbildung 4.9.: Mergesort

```
mergesort(data: Liste): Liste
if length of data = 1
  report data
else
  merge (mergesort (keep items rin-
gify (not (index>round((length of
data)/2))) from data, mergesort( keep
items ringify(index>round((length of
data)/2)) from data))
```

Round((length of data)/2) liefert den Index des mittleren Datensatzes. Für zwei Elemente erhalten wir den Wert 1, für drei den Wert 2. Alle Elemente mit einen größeren Index packen wir in die rechte Hälfte, die übrigen packen wir in die linke Hälfte. Damit ist sichergestellt, dass alle Teilmengen mindestens ein Element haben. Den Größer-gleich-Befehl erhalten wir, indem wir einen Größer-Block einsetzen und dann mittels **relabel** die verborgenen Funktionen aufrufen. Alternativ bilden wir die linke Gruppe, indem wir die Bedingung für die rechte Gruppe `(index>round((length of data)/2))` verneinen. Im Keep-Befehl verwenden wir den Parameter `index`, d.h. die Platznummer der Elemente.

4.6. Quicksort

Ein weit verbreitetes Sortierverfahren ist Quicksort. Die Grundidee ist relativ einfach: Eine Liste wird sortiert, indem wir ein beliebiges Element als Pivotelement auswählen und dann alle Elemente, die kleiner sind als das Pivotelement, in eine Teilliste packen, und alle Elemente, die größer sind, in eine andere Teilliste packen. Die Teillisten werden einzeln mit Quicksort sortiert. Anschließend werden die Teile mit Append zusammengeführt. Dabei ist wichtig, dass das Pivotelement in eine Liste gepackt wird. Als Abbruchbedingung nutzen wir, dass leere Listen und Listen mit einem Element bereits sortiert sind. Im Unterschied zu Mergesort können hier auch leere Teillisten entstehen, wenn wir als Pivotelement das kleinste oder größte Element einer Liste wählen. Ist die Länge der Liste kleiner als zwei, geben wir die Parameterliste zurück. Als Pivotelement wählen wir jeweils das erste Element. Man könnte auch das mittlere Element wählen wie bei Mergesort. Im Keep-Befehl verwenden wir den Index `value`, weil wir die Werte der Elemente mit dem Wert des Pivotelements vergleichen.

```
quicksort(data: Liste): Liste
if (length of data < 2)
  report data
else
  report append(quicksort (keep items
ringify (value<item 1 of data) from
data), list{item 1 of data}, quicksort
(keep items ringify (value>item 1 of
data) from data))
```

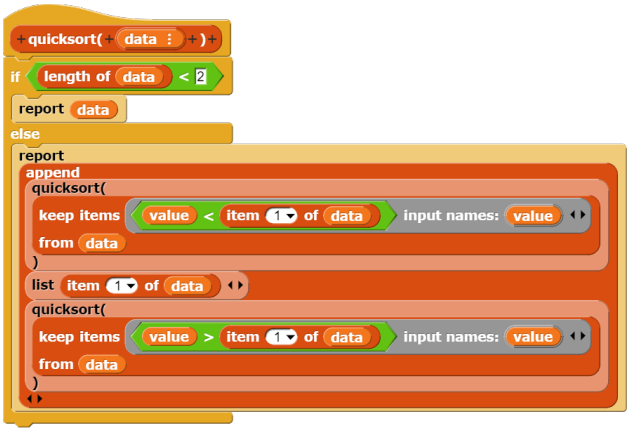


Abbildung 4.10.: Quicksort

Die Programmierung besteht in diesem Fall darin, dass wir die Lösungsidee hinschreiben.

Allerdings hat diese Implementierung noch eine Schwachstelle: Wenn die Liste mehrere gleiche Elemente enthält, werden diese aussortiert, weil die beiden KEEP-Funktionen nur größere oder kleinere Elemente als das Vergleichselement zulassen und gleiche Element heraus filtern. Wir müssen deshalb das „<“-Zeichen durch ein „≤“ersetzen und das erste Element der Liste aus den KEEP-Funktionen ausschließen:

```
quicksort(data: Liste): Liste
if (length of data < 2)
  report data
else
  report append(quicksort (keep items
ringify(value≤item 1 of data) from
all but first of data), list{item 1 of
data}, quicksort(keep items ringify
(value>item 1 of data) from all but
first of data))
```

Ein Tipp: Vergleichen Sie den Aufwand für diese Programmierung mit Funktionen höherer Ordnung mit dem „normalen“ Aufwand für die Programmierung von Quicksort in imperativen Programmiersprachen.

4.7. Aufgaben

Aufgabe 4.1 *Implementieren Sie die behandelten Sortierverfahren.*

Aufgabe 4.2 *Implementieren Sie zwei iterative (z.B. InsertionSort und SelectionSort) und ein rekursives Sortierverfahren (z.B. MergeSort) als Reporter.*

Implementieren Sie eine Operation, die eine Messreihe zeichnet (Siehe Kapitel 2.2)

Fügen Sie die Blöcke zusammen zu einem Programm, das die Laufzeit der verschiedenen Sortierverfahren für Listen verschiedener Länge untersucht und graphisch darstellt. Dabei soll auf der x-Achse die Anzahl der zu sortierenden Elemente und auf der y-Achse die Zeit abgebildet werden.

5. Zweidimensionale Reihungen

In den Abituraufgaben wird der Begriff „Zweidimensionale Reihung“ verwendet. In der Mathematik spricht man von Matrizen (sofern es sich beim Inhalt um Zahlen handelt), umgangssprachlich von Tabellen. In Snap! verwenden wir verkettete Listen, also Listen von Listen zur Darstellung von Tabellen.[1]

In diesem Kapitel lernen Sie

- wie man Tabellen erstellt und am Bildschirm darstellt,
- wie man durch alle Elemente einer Tabelle zeilenweise oder spaltenweise durchgeht,
- wie man eine Kopie einer Tabelle erzeugt,
- wie man eine Umgebung in einer Tabelle bearbeitet,
- wie man mit Tabellen die Programmierung von DEAs vereinfachen kann.

5.1. Listen in Liste

Da Listen in Snap! First-Class-Objekte sind, können sie beliebige andere Objekte als Elemente aufnehmen, natürlich auch andere Listen. Listen in einer anderen Liste ergeben eine Tabelle. In der **Tableview** zeigt Snap! diese Tabelle an.

Dabei werden Elemente, die nicht in einer Liste sind, rot unterlegt dargestellt.

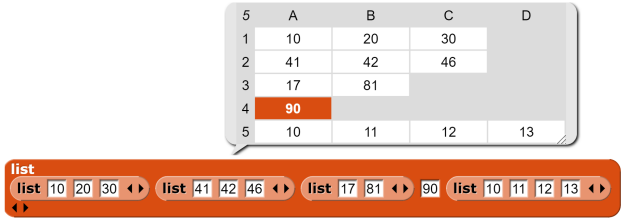


Abbildung 5.1.: Listen in Liste

[1] Eine erste Fassung dieses Kapitels wurde von Mirco Troue erstellt.

In der Regel verwenden wir Listen gleicher Länge für die Zeilen, damit eine Tabelle mit fester Spaltenanzahl (und nicht keine Kammstruktur) entsteht.

5.2. Grundrezept Matrixoperation

Das zeilenweise Durchgehen durch eine Tabelle oder Matrix ist eine der Grundoperationen. Dazu werden zwei ineinander geschachtelte FOR-Schleifen mit zwei Zählvariablen verwendet. In Informatikbüchern werden dafür in der Regel die Buchstaben i und j eingesetzt. Sicherer ist es aber, wenn man s (für Spalte) und z (für Zeile) verwendet.

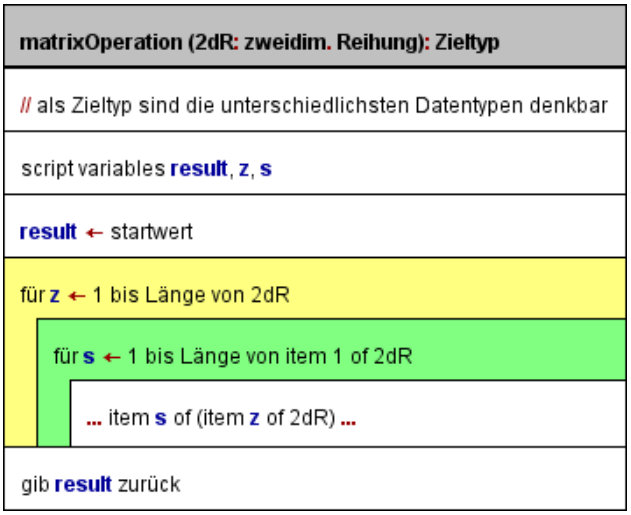


Abbildung 5.2.: Grundrezept Matrixoperation

Im folgenden Beispiel wird eine zweidimensionale Reihung namens 2dR vorausgesetzt:

```
for z = 1 to length of 2dR
  for s=1 to length of(item 1 of 2dR)
    ... item s of (item z of 2dR) ...
```

Der Zugriff auf die einzelnen Elemente der Tabelle erfolgt immer von innen nach außen, also `item spalte of (item zeile of tabelle)`.

`Variation`: Um die Tabelle spaltenweise durchzugehen, werden die beiden FOR-Schleifen vertauscht.

Beispiel: Der folgende Block liefert die Zeilensummen einer Tabelle:

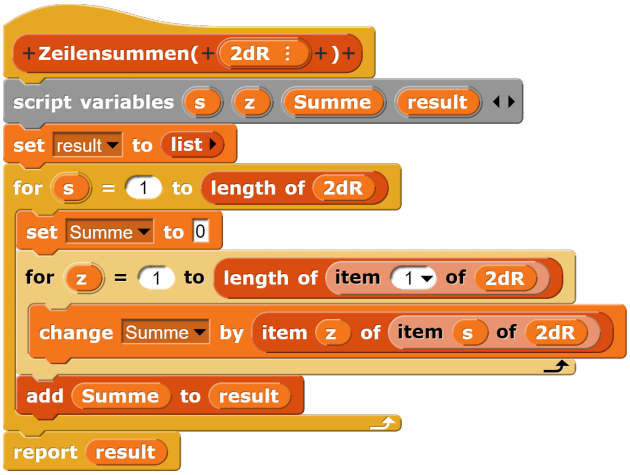


Abbildung 5.3.: Zeilensummen

Vertauscht man die beiden `FOR`-Schleifen, so erhält man die Spaltensummen der Tabelle:

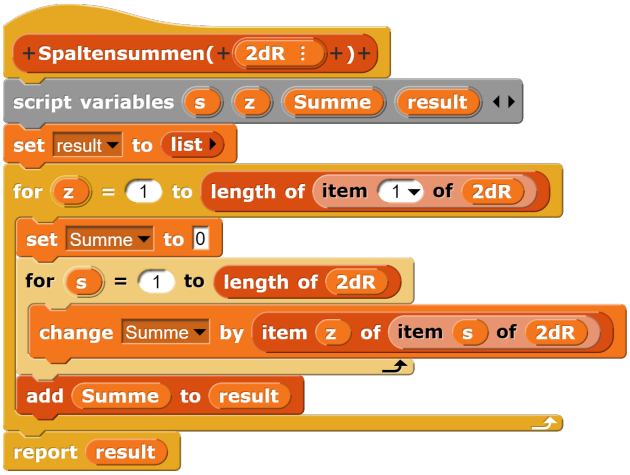


Abbildung 5.4.: Spaltensummen

5.3. Kopieren von Tabellen

Bei der Behandlung der Listen haben wir festgestellt, dass sich Listen nicht durch einen einfachen SET-Befehl kopieren lassen, weil dann nur die Referenz kopiert wird und Original und Kopie an der gleichen Stelle im Speicher stehen, also Änderungen der einen Liste sich sofort auf die andere Liste auswirken. Wir erstellen das folgende Skript und testen es:

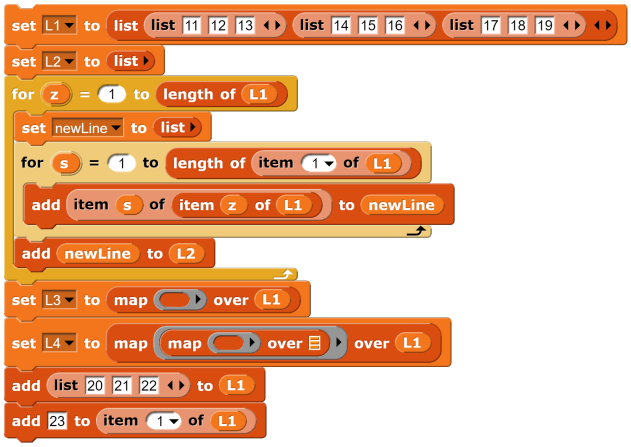


Abbildung 5.5.: Kopieren von Tabellen

Zunächst wird eine Tabelle L1 erstellt. Von dieser Tabelle werden dann auf drei verschiedene Arten Kopien erstellt: L2 entsteht, indem die neue Tabelle Element für Element aus der alten zusammengesetzt wird. Dafür verwenden wir zwei geschachtelte `FOR`-Schleifen. L3 entsteht durch die Verwendung des `MAP`-Befehls. L4 entsteht durch eine geschachtelte Verwendung des `MAP`-Befehls. Wir testen das Skript bis hierhin und stellen fest, dass es drei identische Kopien erzeugt. Dann nehmen wir zwei Veränderungen an L1 vor: Wir ergänzen eine vierte Zeile und hängen in der ersten Zeile ein Element an. Anschließend prüfen wir die Auswirkungen auf die anderen Listen: Eine vierte Zeile gibt es in keiner der drei anderen Tabellen. Allerdings wurde in L3 in der ersten Zeile ein Element angehängt.

Wir halten fest: Ein Elemente weises Kopieren mit zwei `FOR`-Schleifen erzeugt eine echte Kopie einer Tabelle. Ein einfaches `MAP]` führt zwar zu einer Kopie der Zeilen, wobei die Zeilen selbst aber als Referenzen gespeichert werden. Deshalb werden keine neuen Zeilen angefügt, wenn in der Ursprungstabelle Zeilen angefügt werden. Aber die einzelnen Zeilen werden verändert, wenn wir sie in der Ursprungstabelle verändern. Eine echte Kopie einer Tabelle entsteht also nur durch die geschachtelte `FOR`-Schleife oder eine geschachtelte `MAP`-Anweisung, `set kopie to map ringify(map ringify([ ]) over [ ]) over original`.

5.4. Vergleich der Elemente einer Liste

Eine Variation des Grundrezepts Matrizenoperation kann auch dazu verwendet werden, die Element einer Liste miteinander zu vergleichen. Dazu können wir vier Fälle unterscheiden:

1.					3.				
*	*	*	*	*	*	*	*	*	*
*	*	*	*	*	_	*	*	*	*
*	*	*	*	*	_	_	*	*	*
*	*	*	*	*	_	_	_	*	*
*	*	*	*	*	_	_	_	_	*
2.					4.				
_	*	*	*	*	_	*	*	*	*
*	_	*	*	*	_	_	*	*	*
*	*	_	*	*	_	_	_	*	*
*	*	*	_	*	_	_	_	_	*
*	*	*	*	_	_	_	_	_	_

Tabelle 5.1.: Vergleich der Elemente einer Liste

1. die Vergleichsoperation ist nicht kommutativ (d.h. vergleiche (a,b) gibt nicht dasselbe Ergebnis wie vergleiche (b,a) und der Selbstvergleich wird eingeschlossen. Dann lautet die Vergleichsoperation:

```
for i=1 to length of data
  for j=1 to length of data
   vergleiche(item 1 of data, item
j of data)
```

2. die Vergleichsoperation ist nicht kommutativ, aber der Selbstvergleich eines Elements mit sich selbst wird ausgeschlossen. Dann lautet die Vergleichsoperation:

```
for i=1 to length of data
  for j=1 to length of data
    if not (i=j)
      vergleiche(item 1 of data,
      item j of data)
```

3. die Vergleichsoperation ist kommutativ, aber der Selbstvergleich eines Elements mit sich selbst ist nicht ausgeschlossen. Der zweite Index kann dann bei i beginnen. Dann lautet die Vergleichsoperation:

```
for i=1 to length of data
  for j=i to length of data
    vergleiche(item 1 of data,
    item j of data)
```

   Ein Beispiel wäre die Überprüfung eines Codes auf Linearität, bei der alle Kombinationen von Elementen gebildet werden müssen.

4. die Vergleichsoperation ist kommutativ und der Selbstvergleich eine Elements ist ausgeschlossen. Die erste Zählvariable kann dann die beiden letzten Zeilen auslassen (die letzte Zeile ist leer und die vorletzte enthält nur ein Element) und die zweite Zählvariable kann bei i+1 beginnen. Dann lautet die Vergleichsoperation:

```
for i=1 to (length of data −2 )
  for j=i+1 to length of data
    vergleiche(item 1 of data,
    item j of data
```

   Ein Beispiel wäre die Berechnung des Hammingabstandes für einen Code.

5.5. Umgebungen in Tabellen

In manchen Aufgaben werden nicht ganze Tabellen betrachtet, sondern nur Umgebungen einzelner Elemente. Gegeben ist sei ein Bild in Form einer zweidimensionalen Reihung (Tabelle). Jedes Element der Tabelle ist ein Bildpunkt. Die Farben sind als Zahlen codiert. Gesucht ist die Summe der acht Nachbarfelder zu einen durch Zeile und Spalte festgelegten Element der Tabelle. Zunächst lassen wir die Randfelder außer Betracht.

Zeile und Spalte bestimmen hier das Feld, dessen Umgebung betrachtet wird, s und z werden wir als Zählvariablen, um das aktuelle Feld zu beschreiben.

Wir verwenden wie gewohnt zwei geschachtelte `FOR`-Schleifen, um die einzelnen Elemente erfassen. Dabei laufen die Zeilen von (`Zeile`−1) bis (`Zeile`+1) und die Spalten von

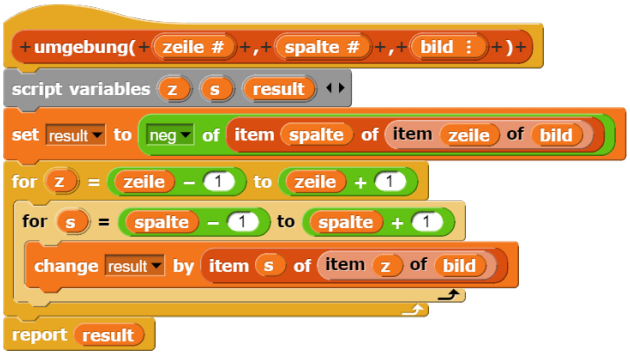


Abbildung 5.6.: Summe der acht umgebenden Felder

(`Spalte−1`) bis (`Spalte+1`). Damit sind nicht nur die acht Felder der Umgebung erfasst, sondern auch das Element (Zeile|Spalte). Um dessen Einfluss auszuschalten, initialisieren wir `result` mit der Gegenzahl dieses Elements. Alternativ könnte man auch eine Bedingung in die beiden `FOR`-Schleifen einschließen: `if not (s=spalte and z=zeile)`.

Wie müssen wir den Block ergänzen, um auch die Randfelder richtig zu berechnen? Dazu überlegen wir zunächst, welche Felder gültig sind. Gültig sind alle Felder mit einem Zeilenwert zwischen 1 und der Länge des Bildes und einem Spaltenwert zwischen 1 und der Länge des ersten Elements (der ersten Zeile) des Bildes. Diese Bedingungen können wir wie folgt zusammenfassen: **(z>0 and z<Länge des Bildes +1 and s>1 and s<Länge des ersten Elements des Bildes+1)**

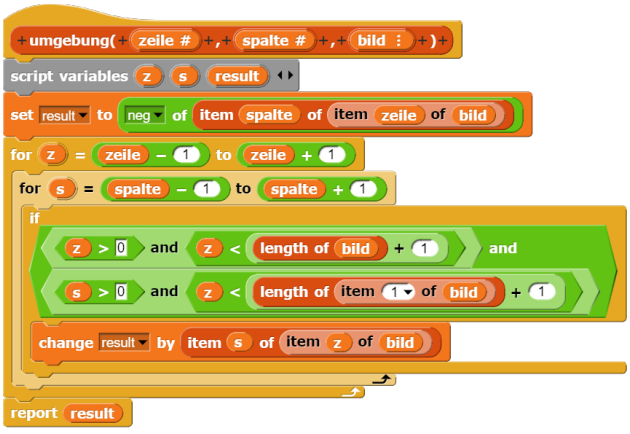


Abbildung 5.7.: Summe der acht umgebenden Felder für Randfelder

Damit können wir auch die Randfelder richtig erfassen.

5.6. Programmierung von endlichen Automaten

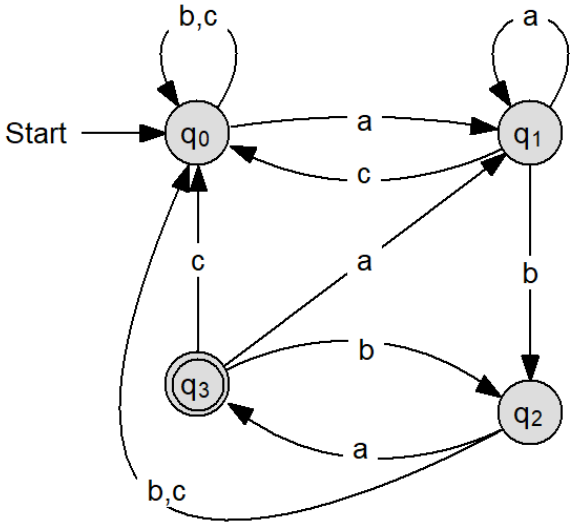


Abbildung 5.8.: DEA für „endet auf aba“

Als Beispiel betrachten wir den endlichen Automaten mit dem Eingabealphabet {a, b, c}, der alle Worte erkennt, die auf „aba“ enden.

5.6.1. 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.

`Delta` besteht überwiegend aus geschachtelten Verzweigungen. Zunächst müssen die vier Zustände unterschieden werden und dann das jeweilige Eingabezeichen.

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 eine Reihe globaler Variablen: das Eingabealphabet, die Zustandsmenge, die Menge der akzeptierenden Zustände, den (aktuellen) Zustand, das Wort, das überprüft werden soll, und eine Zählvariable i, weil wir das Wort Zeichen weise durchgehen müssen. Das Hauptprogramm setzt den Zustand q0 (Startzustand),

Abbildung 5.9.: Überführungsfunktion als geschachtelte Schleife

Abbildung 5.10.: Eingabe-Block erste Fassung

und das Wort auf die eingegebene Zeichenfolge. Dann wird für jeden Buchstaben `letter i of wort` aus Zustand und Eingabezeichen der Folgezustand bestimmt. Zum Schluss überprüft das Programm, ob der aktuelle Zustand in der List der akzeptierenden Zustände enthalten ist und macht eine entsprechende Ausgabe:

In der aktuellen Fassung unseres DEA greifen wir noch nicht auf das Eingabealphabet und die Liste der Zustände zurück.

5.6.2. Erste Verbesserung: Filterung der Eingabe

In der jetzigen Fassung ist der Automat nicht gegen Fehleingaben gesichert. Z.B. wird das Wort „a2b“ akzeptiert, obwohl 2 nicht zu den gültigen Eingabezeichen gehört. Nun könnte man in die Eingabe eine Abfrage einbauen, ob das Wort nur aus den Buchstaben a, b und c

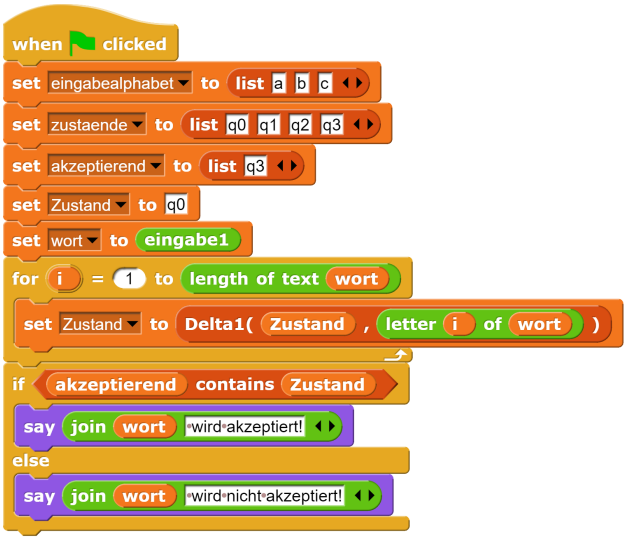


Abbildung 5.11.: Hauptskript

besteht. Wir wählen hier einen anderen Weg, der sich stärker an der Definition des DEA orientiert. 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.

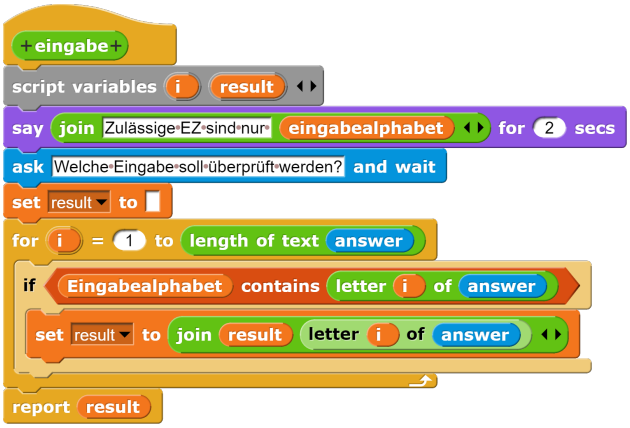


Abbildung 5.12.: Eingabe-Block verbesserte Fassung

5.6.3. Zweite Verbesserung: 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/Flaci.com als Tabelle:

δ	a	b	c
q_0	q_1	q_0	q_0
q_1	q_1	q_2	q_0
q_2	q_3	q_0	q_0
q_3	q_1	q_2	q_0

Tabelle 5.2.: Überführungsfunktion DEA

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

Dazu benötigen wir den Index, an denen der Zustand in der Zustandsmenge und das Eingabezeichen im Eingabealphabet stehen. Dafür gibt es seit Snap! 6 den Block `index of element in liste`:

Abbildung 5.13.: Neue Fassung der Überführungsfunktion

Im Hauptskript benötigen wir eine zusätzliche globale Variable `delta`, in der wir die Tabelle der Überführungsfunktion speichern:

Diese Version des DEA lässt sich problemlos an einen anderen endlichen Automaten anpassen. Wir müssen lediglich das Eingabealphabet, die Zustandsmenge und die Menge der akzeptierenden Zustände anpassen und die Tabelle mit der neuen Überführungsfunktion einsetzen.

5.6.4. Dritte Verbesserung: Funktionen höherer Ordnung

Das Filtern der Eingabe lässt sich durch Verwendung der `Keep`-Funktion viel eleganter gestalten. Wir wandeln die Eingabe durch `split`

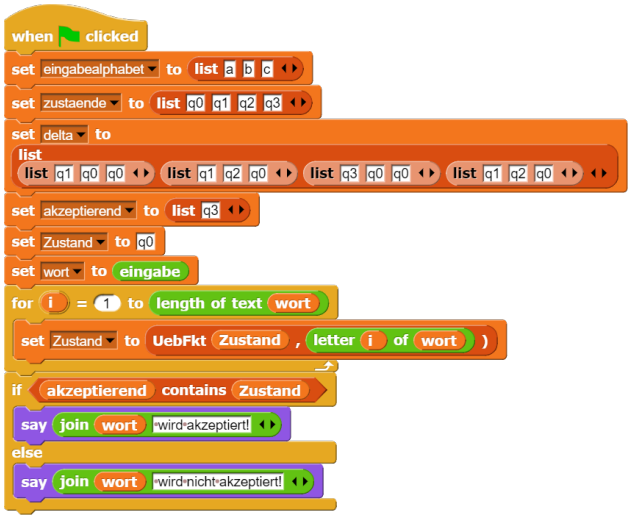


Abbildung 5.14.: Neue Version des DEA

`Wort by letter` in eine Liste um und behalten nur die Elemente, die im Eingabealphabet enthalten sind:

Abbildung 5.15.: Filtern der Eingabe

Unsere Überführungsfunktion besteht ja nur aus einer einzelnen Zeile. Insofern ist die Funktion eigentlich überflüssig (zumal sie nur an einer Stelle aufgerufen wird). Wir können die geschachtelten `item of Index in Liste`-Blöcke auch direkt in die Zuweisung einsetzen.

Die `FOR`-Schleife können wir durch einen `for each item in Liste`-Block ersetzen, wobei wir dann in der Zuweisung `item` statt `EZ` als Spaltenindex verwenden müssen.

Damit erhalten wir die dritte Fassung unseres deterministischen endlichen Automaten:

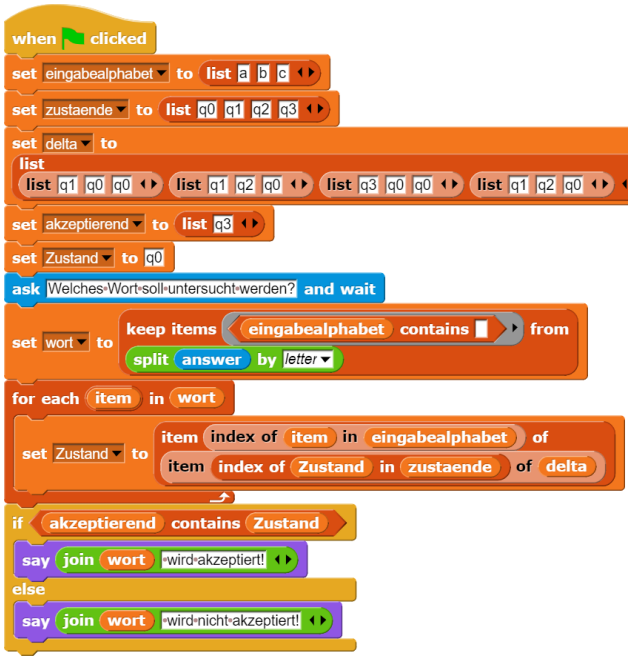


Abbildung 5.16.: 3. Version des DEA

5.7. Aufgaben

Aufgabe 5.1 *Implementiere eine Operation* `Summe`*, die die Summe aller Elemente einer zweidimensionalen Reihung ausgibt.*

Aufgabe 5.2 *Implementiere eine Operation* `Anzahl`*, die die Anzahl der Elemente einer zweidimensionalen Reihung ausgibt.*

Aufgabe 5.3 *Implementiere eine Operation* `Mittelwert`*, die den Mittelwert der Zahlen in einer zweidimensionalen Reihung ausgibt.*

Aufgabe 5.4 *Implementiere eine Operation* `Zeilensumme`*, die eine Liste mit den Summen der einzelnen Zeilen einer zweidimensionalen Reihung ausgibt.*

Aufgabe 5.5 *Implementiere eine Operation* `Spaltensumme`*, die eine Liste mit den Summen der einzelnen Spalten einer zweidimensionalen Reihung ausgibt.*

Aufgabe 5.6 *Implementiere eine Operation* `Umrahmung`*, die an eine zweidimensionale Reihung die letzte Spalte vor der ersten und die erste Spalte nach der letzten sowie die erste Zeile nach der letzten und die letzte Spalte vor der ersten einfügt und die „umrahmte“ Reihung ausgibt.*

1	2	3
4	5	6
7	8	9

9	7	8	9	7
3	1	2	3	1
6	4	5	6	4
9	7	8	9	7
3	1	2	3	1

Abbildung 5.17.: Umrahmung einer 3x3-Reihung

Beispiel: Aus einer 3x3-Reihung wird eine 5x5-Reihung:

Aufgabe 5.7 *Implementiere eine Operation* `Umgebungsumme`*, die zu einer zweidimensionalen Matrix und zu einer gültigen Spalte und Zeile die Summe der acht umgebenden Reihung ausgibt – ohne Berücksichtigung von Randfeldern.*

Ergänze die Implementierung von Umgebungssumme, so dass auch Rand- und Eckfelder als Eingabe möglich sind. Beispiel: Für jedes Eckfeld soll die Summe der drei angrenzenden Felder ausgegeben werden.

Aufgabe 5.8 *Implementiere einen Block, der das kleinste Element einer zweidimensionalen Reihung liefert.*

Aufgabe 5.9 *Implementiere einen Block, der die drei größten Elemente einer zweidimensionalen Reihung liefert.*

Aufgabe 5.10 *Implementiere einen Block* `enthalten` *mit den Parametern* `bild` *- als zweidimensionale Reihung - und* `grenzwert`*, der angibt, ob Bildpunkte im Bild enthalten sind, deren Zahlenwert größer ist als der Grenzwert.*

Aufgabe 5.11 *Implementiere einen Block* `umwandeln` *mit den Parametern* `bild` *- als zweidimensionale Reihung - und* `schranke`*, der ein Bild in Grautönen in ein Schwarz-Weiß-Bild umwandelt. Dazu sollen alle Bildpunkte oberhalb der Schranke auf 1 gesetzt werden und alle anderen Bildpunkte auf 0. Die Grautöne ist gegeben als Zahlen zwischen 0 (weiß) und 255 (schwarz).*

Aufgabe 5.12 *Gib eine Tabelle mit dem kleinen Einmaleins aus.*

Aufgabe 5.13 *Eine zweidimensionale Reihung* `Wahlergebnis` *enthält in den Zeilen die Anzahl der gültigen Stimmen, die in jedem Wahlbezirk abgegeben wurden, und in den Spalten die Anzahl der Stimmen, die für jede Partei abgegeben wurde. Das Feld in der dritten Spalte der zweiten Zeile enthält also die Anzahl der Stimmen, die im Wahlbezirk 2 für Partei 3 abgegeben wurden.*

Implementiere eine Operation `Stimmen`*, die die Anzahl der Stimmen für jede Partei ausgibt.*

Implementiere darauf aufbauend eine Operation `Prozente`*, die den prozentualen Anteil jeder Partei ausgibt, gerundet auf eine Nachkommastelle (z.B. 17,4%).*

Aufgabe 5.14 *In der Formel-1-Saison 2000 starteten 22 Fahrer in insgesamt 17 Rennen. Die Punkte für die Fahrer-Weltmeisterschaft werden folgendermaßen verteilt: Erreichte ein Fahrer in einem Rennen den ersten Platz, so erhielt er dafür 10 Wertungspunkte, für den 2. bis 6. Platz erhielt er 6, 4, 3, 2 bzw. 1 Punkt. In allen anderen Fällen bekam der Fahrer keinen Punkt.*

Eine Tabelle `WM2000` *mit der Übersicht über die gesamten Ergebnisse einer Rennsaison enthält in Form einer zweidimensionalen Reihung in den Zeilen die Ergebnisse der einzelnen Rennen und in den Spalten die jeweilige Platzierung der Fahrer. Dabei bedeutet Platzierung 0, dass der Fahrer in einem Rennen das Ziel nicht erreicht hat.*

Implementieren Sie einen Algorithmus, der die Gesamtpunktzahl jedes Fahrers berechnet und eine Liste `PunkteWM2000` *einträgt.*

Implementieren Sie darauf aufbauend einen Algorithmus, der den Weltmeister bestimmt. Beachten Sie, dass bei Punktgleichheit zweier Fahrer derjenige der „bessere“ ist, der öfter den ersten Platz in einem Rennen erreicht hat.

Aufgabe 5.15 *(Game of Life) Der Zustandes eines zellulären Automaten wird durch eine zweidimensionale Reihung* `Life` *mit den Dimensionen* `hoehe` *und* `breite` *bestimmt. Dabei steht 0 für eine tote Zelle und 1 für eine lebende Zelle.*

- *Implementiere eine Operation* `livingCells`*, die angibt, wie viele lebende Zellen* `Life` *enthält.*
- *Implementiere eine Operation* `nachbarn`*, die zu einer Zelle angibt, wie viele der acht Nachbarfelder mit einer lebenden Zelle belegt sind.*
- *Implementiere eine Operation* `nextGeneration`*, die als Parameter die Reihung* `Life` *erhält und den neuen Zustand des Automaten nach folgenden Regeln ausgibt:*

  1. Eine tote Zelle mit genau 3 lebenden Nachbarn wird in der nächsten Generation neu geboren.

  2. Lebende Zellen mit weniger als zwei lebenden Nachbarn sterben in der nächsten Generation an Einsamkeit.

  3. Lebende Zellen mit zwei oder drei lebenden Nachbarn bleiben am Leben.

  4. Lebende Zellen mit mehr als drei lebenden Nachbarn sterben in der nächsten Generation an Überbevölkerung.

- *Die Operation* `Vergleich3x3` *erhält als Parameter ein beliebiges Muster von 3x3 Zellen sowie Zeile und Spalte. Das Muster wird durch eine Zeichenkette beschrieben.*

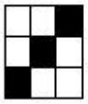


Abbildung 5.18.: Muster 001010100

Beispiel: Das Muster unten wird als 001010100 übergeben. Implementieren Sie die Operation `Vergleich3x3`*, die überprüft, ob sich in dem 3x3 Zellen großen Quadrat, dessen linke obere Ecke durch Zeile und Spalte beschrieben wird, lebende Zellen in Form des Musters befinden.*

- *Ergänzen Sie das Game of Life um eine Operation zur graphischen Ausgabe des aktuellen Zustands.*

6. Stapel, Schlange, dynamische Reihung

In diesem Kapitel lernen Sie

- was wir unter den Klassen Stapel und Schlange verstehen und wie diese Klassen implementiert werden.
- wie wir die Klassen Stapel und Schlange verwenden.
- was der Unterschied zwischen einer Snap!-Liste und einer dynamischen Reihung ist.

6.1. Klasse Stapel

Ein Stapel (englisch **Stack** ist eine dynamische Datenstruktur, die von der Entstehung zwischen den statischen Reihungen und den dynamischen Listen (vgl. Kapitel 1.2.) steht. Dynamisch bedeutet in diesem Zusammenhang, dass weder die Größe der einzelnen Elemente noch ihre Anzahl festgelegt ist. Ein Stapel hat wie die dynamische Liste eine Referenz (einen Anker) im Stack, die auf das erste Element zeigt. Am Ende des Element zeigt eine andere Referenz auf das nächste Element usw.

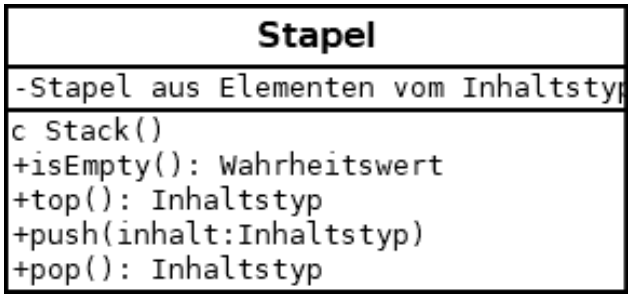


Abbildung 6.1.: Klasse Stapel

Anschaulich kann man sich einen Stapel vorstellen wie einen Stapel Pfannkuchen. Man kann einen Pfannkuchen auf dem Stapel ablegen oder von oben wegnehmen. Es ist aber nicht möglich, einen Pfannkuchen von weiter unten aus dem Stapel zu ziehen (es sei denn, man hebt einen Teil des Stapels vorher ab). Man kann auch die Oberseite des obersten Pfannkuchens betrachten oder nachsehen, ob der Pfannkuchenstapel leer ist. Ein Bücherstapel ist keine so gute Analogie, weil man aus einem Bücherstapel auch ein Buch herausziehen kann, das weiter unten liegt. Das geht bei den Pfannkuchen nicht. Der Stapel funktioniert nach dem **LIFO-Prinzip: Last In, First Out**.

Wichtig ist bei der Klasse Stapel wie bei den folgenden Klassen das **Geheimnisprinzip** und die **Datenkapselung**. Als Datenkapselung bezeichnet man in der Informatik das Verbergen von Daten vor dem Zugriff von außen. Direkte Zugriffe auf die Datenstruktur werden unterbunden. Nur die definierten Zugriffsoperationen sind zulässig. Datenkapselung ist eine Umsetzung des Geheimnisprinzips, um unbeabsichtigte Seiteneffekte zu vermeiden.

Obwohl wir - mangels oder dank des Fehlens von Pointern in Snap! - den Stapel mit einer Snap!-Liste modellieren, dürfen wir auf Stapel ausschließlich die Stapeloperationen anwenden und **nicht** Listenoperationen wie `length of` oder `contains`.

Für die Implementierung eines Stapels mit Hilfe einer Snap!-Liste gibt es verschiedene Möglichkeiten. Wir wählen hier eine Liste mit zwei Elementen, wobei das erste Element den Wert „Stapel“ hat und das zweie Element eine Liste mit den Stapelelementen ist. Diese Implementation hat den Vorteil, dass Listenoperationen nicht ohne weiteres funktionieren. Alternativ könnte man auch eine einfache Snap!-Liste verwenden oder die Typbezeichnung in das erste Element schreiben.

Es gibt insgesamt fünf Stapel-Operationen: `Stack()` ist ein Konstruktor. `Set myStapel to Stack()` erzeugt einen neuen Stapel und legt ihn unter dem Namen myStapel ab. Alternativ können wir im Quelltext auch schreiben: `myStapel.Stack()`.

`myStapel.isEmpty(): Wahrheitswert` ist ein Predicate, das angibt, ob der Stapel leer

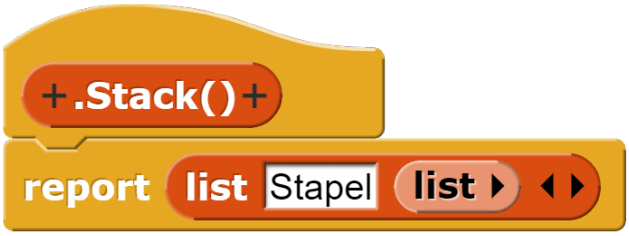


Abbildung 6.2.: Konstruktor Stack

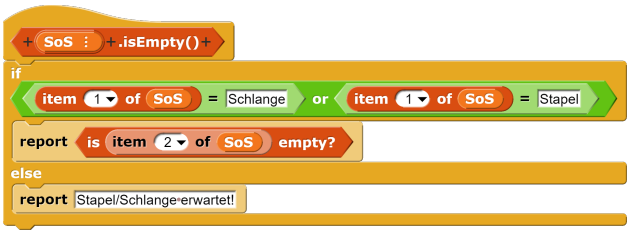


Abbildung 6.3.: Predicate isEmpty

ist. Da die Schlange ebenfalls über eine Operation `isEmpty()` verfügt, implementieren wir die Operation so, dass sowohl leere Schlangen wie auch leere Stapel erkannt werden.

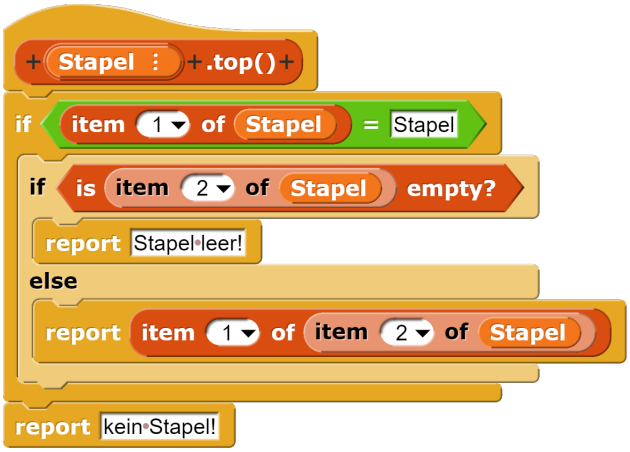


Abbildung 6.4.: Reporter top()

`myStapel.top(): Inhaltstyp` ist ein Reporter, der den Inhalt des obersten Elements anzeigt, ohne dieses zu verändern.

`myStapel.push(inhalt: Inhaltstyp` ist ein Command, das ein neues Element oben auf dem Stapel ablegt.

`myStapel.pop():Inhaltstyp` ist ein Reporter, der den Inhalt des obersten Elements des Stapels liefert und dieses Element gleichzeitig aus dem Stapel entnimmt.

Alle Operationen der Klasse Stapel sind in den „Erläuterungen zu Operationen und Datenbankabfragen“ enthalten,

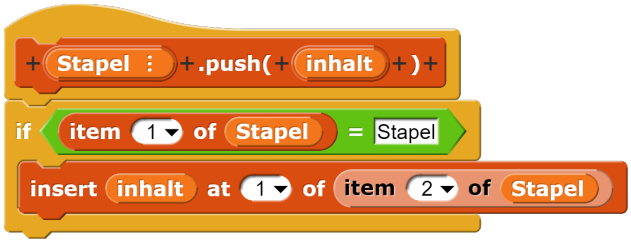


Abbildung 6.5.: Command push(inhalt)

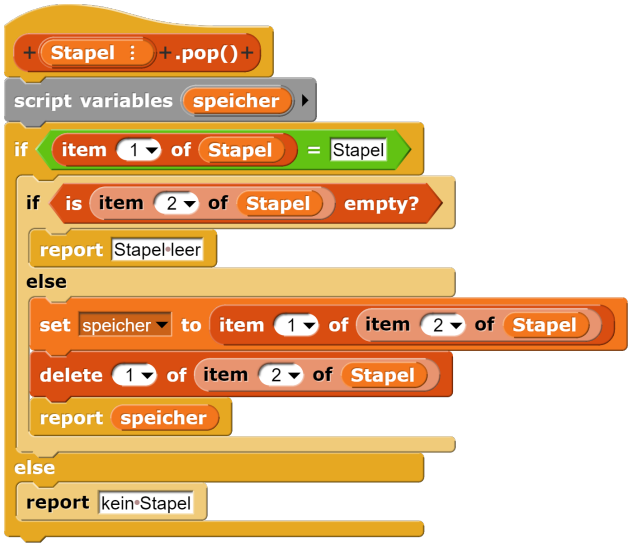


Abbildung 6.6.: Reporter pop()

die Ihnen in der Abiturprüfung schriftlich vorliegen. Es ist erstaunlich, wie viele Schülerinnen und Schüler trotzdem falsche Operationen verwenden, wie z.B. `item i of Liste`. Bitte gewöhnen Sie es sich an, dass Sie das Blatt mit den Operationen neben sich legen, wenn Sie mit Stapeln programmieren.

6.2. Klasse Schlange

Schlange
-Schlange aus Elementen vom Inhaltstyp
c Queue() +isEmpty(): Wahrheitswert +head(): Inhaltstyp +enqueue(inhalt:Inhaltstyp) +dequeue(): Inhaltstyp

Abbildung 6.7.: Klasse Schlange

Eine Schlange funktioniert ähnlich wie ein Stapel, nur dass man sich bei der Schlange hinten

anstellen muss und dann langsam nach vorn vorrückt. Eine Schlange funktioniert also nach dem **FIFO-Prinzip: First In, First Out**, d.h. das Element, das zuerst da war, wird auch zuerst behandelt.

Auch bei einer Schlange gelten Datenkapselung und Geheimnisprinzip. Die Möglichkeiten der Implementierung sind die gleichen wie beim Stapel.

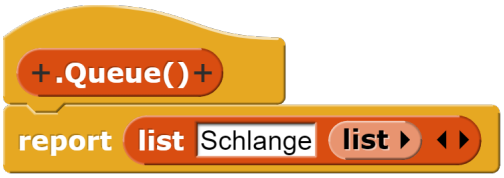


Abbildung 6.8.: Konstruktor Queue()

Der Konstruktor `Queue()` erzeugt eine neue Schlange. In unserer Implementation speichern wir im ersten Element der Liste die Typbezeichnung „Schlange“, das zweite Element enthält eine Liste mit dem Inhalt der Schlange. Wir notieren `set mySchlange to Queue()` oder `mySchlange.Queue()`.

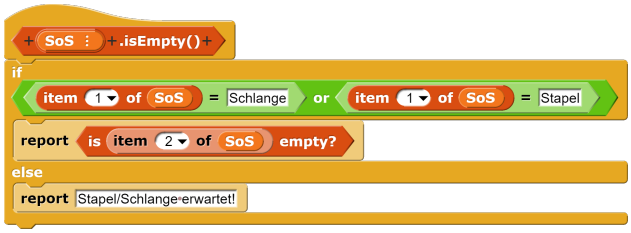


Abbildung 6.9.: Predicate isEmpty()

Das Predicate `isEmpty()` gibt an, wo die Schlange leer ist.

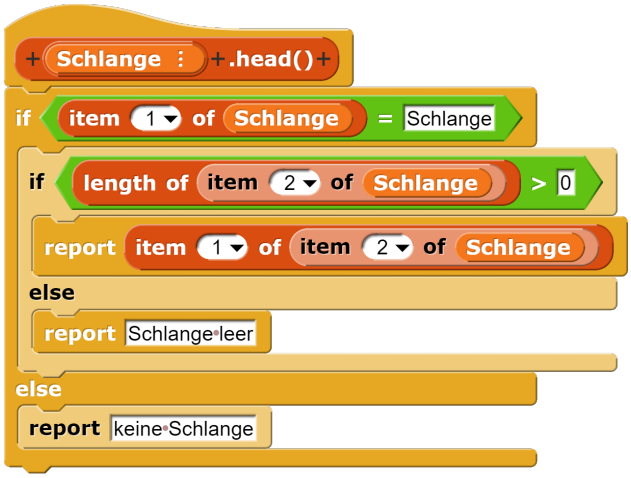


Abbildung 6.10.: Reporter head()

Der Reporter `head()` liefert das vorderste Element der Schlange.

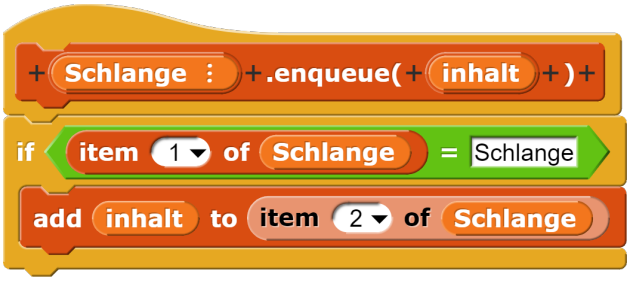


Abbildung 6.11.: Command enqueue()

Das Command `enqueue(inhalt)` fügt ein neues Element hinten an die Schlange an.

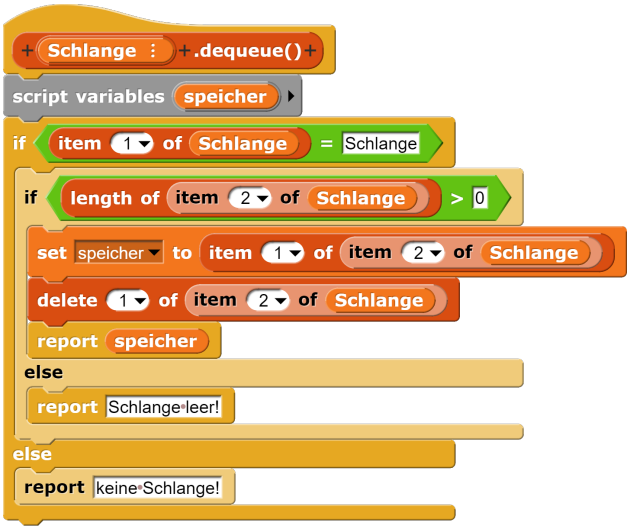


Abbildung 6.12.: Reporter dequeue()

Der Reporter `dequeue()` liefert das erste Element der Schlange und löscht gleichzeitig dieses Element aus der Schlange.

Alle Operationen der Klasse Schlange sind in den „Erläuterungen zu Operationen und Datenbankabfragen“ enthalten, die Ihnen in der Abiturprüfung schriftlich vorliegen. Es ist erstaunlich, wie viele Schülerinnen und Schüler trotzdem falsche Operationen verwenden, wie z.B. item i of ... . Bitte gewöhnen Sie es sich an, dass Sie das Blatt mit den Operationen neben sich legen, wenn Sie mit Schlangen programmieren.

6.3. Klasse Dynamische Reihung

Seit der Abiturprüfung 2018 ist an die Stelle der Klasse `Liste` die Klasse Dynamische Reihung getreten. Die Klasse Liste gestattete zwar den Zugriff auf beliebige Listenelemente, aber erst nachdem der interne Positionszeiger mit `positionSetzen(position)` entsprechend gesetzt wurde. Die neue Klasse Dynamische Reihung ist viel stärker nach dem Vorbild der indizierten Listen in Snap! modelliert. Der Vorteil, den die Benutzer von Snap! bisher dadurch hatten, daß sie über den Index auf jedes einzelne Element zugreifen können, wird dadurch nivelliert.

Allerdings beginnt die Zählung der Elemente der dynamischen Reihung - wie bei den statischen Reihungen aus der Frühzeit der Programmierung) nicht mit 1, sondern mit 0. Das erste Element ist also das 0., das zweite das 1., das n-te Element trägt die laufende Nummer n-1.

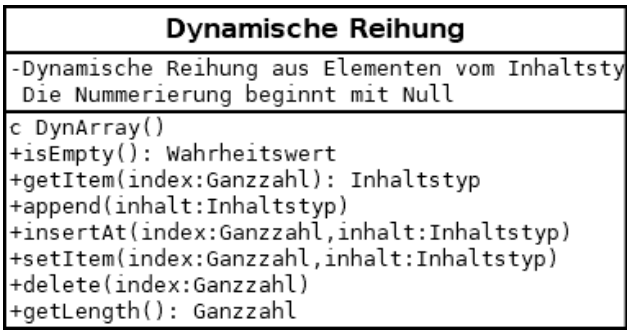


Abbildung 6.13.: Objekt Dynamische Reihung

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

Das Prädikat `isEmpty(): Wahrheitswert` heißt in Snap! `is Liste empty?`.

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

`Append(inhalt: Inhaltstyp)` wird durch `add inhalt do Liste` realisiert. **Achtung**, hier besteht Verwechslungsgefahr! In Snap! steht `append Liste1, Liste2, ...` für das Zusammenfügen von Listen, nicht für das Hinzufügen eines Elements zu einer Liste.

Das `getLength()` der Dynamischen Reihung heißt `length of Liste` bei Snap!-Listen.

Die Methode `insert(index: Ganzzahl, inhalt: Inhaltstyp)` wird von `insert inhalt at index of Liste` übernommen. Beide Operationen funktionieren sowohl innerhalb der Liste als auch an der Stelle direkt hinter dem letzten Listenelement. Die Snap!-Version funktioniert auch mit Indices, die größer sind als die Länge der Liste (inhalt wird dann an das Ende der Liste angefügt). Die DynArray-Version produziert für diesen Fall einen Laufzeitfehler, der durch entsprechende Abfragen explizit ausgeschlossen werden muss.

Neu ist `setItem(index: Ganzzahl, inhalt: Inhaltstyp)`, das dem `replace item index of Liste with inhalt` entspricht. Auch hier gilt: Die DynArray-Operation von `setItem` setzt voraus, dass der Index auf ein existierendes Element der Liste zeigt. Andernfalls wird ein Laufzeitfehler produziert. `Replace` ergänzt die Liste um leere Elemente, wenn der Index größer ist als die Länge der Liste, und fügt dann den angegebenen Inhalt hinten an.

`Delete(index: Ganzzahl)` löscht das Element an der angegebenen Stelle. `Delete index of Liste` in Snap! hat die gleiche Wirkung. Zeigt der Index bei der Snap!-Operation auf ein nicht existierendes Element, wird der Fehler abgefangen. Die DynArray-Operation erzeugt einen Laufzeitfehler.

`GetLength(): Ganzzahl` liefert die Länge der Liste ebenso wie `length of Liste`.

Der Vorteil bei der Verwendung des Datentyps Dynamische Reihung besteht darin, dass mit dem Hilfsmittel für Prüflinge zur Verwendung in der schriftlichen Abiturprüfung im Fach Informatik die Operationen in Papierform vorliegen. Der Nachteil liegt in der ungewohnten Zählung der Elemente beginnend mit Null und in der Möglichkeit von Laufzeitfehlern, die bei der Verwendung einer Snap!-Liste automatisch abgefangen werden.

6.4. Aufgaben

Aufgabe 6.1 *Gegeben ist ein Stapel, der Zufallszahlen enthält. Implementiere eine Operation* `toList`*, die den Stapel als Parameter erhält und die eine sortierte Liste ausgibt.*

Aufgabe 6.2 *Gegeben sei eine Liste* `Namensliste`*, die Namen enthält. Implementiere eine Operation* `List2Schlange`*, der die Namensliste übergeben wird und die eine alphabetisch sortierte Schlange zurückgibt.*

Aufgabe 6.3 *Gegeben seien zwei Stapel* `Stack1` *und* `Stack2`*, die Zeichenketten enthalten.*

a) *Implementiere eine Operation* `Merge1`*, der beide Stapel übergeben werden und die die Zeichenketten abwechselnd in einen neuen Stapel einfügt, der dann zurückgegeben wird. Beachten Sie, dass die Stapel unterschiedlich viele Elemente enthalten können.*

b) *Ergänzen Sie die Operation* `Merge1` *so, dass Umlaute richtig eingefügt werden (Ä=AE, Ö=OE, usw.).*

Aufgabe 6.4 *Gegeben seien zwei sortierte Schlangen namens* `Heap1` *und* `Heap2`*, die Zahlen enthalten. Implementiere eine Operation* `Merge2`*, der* `Heap1` *und* `Heap2` *übergeben werden und die eine sortierte Schlange liefert.*

Aufgabe 6.5 *Ein Stapel namens Kartenspiel enthält Spielkarten. Implementiere eine Operation* `Mische` *zum Mischen der Spielkarten.*

Aufgabe 6.6 *Während der Schweinegrippeepidemie wurden in einigen Ländern an der Flughafen-Einreisekontrolle Wärmebildkameras aufgestellt, um Personen mit erhöhter Temperatur zu identifizieren. Wärmebilder sind oft grob gerastert, stellen aber die Temperatur der gezeigten Bereiche in unterschiedlichen Farben dar. Die benutzte Wärmebildkamera liefert Grauwerte aus dem Bereich von 0 (Schwarz) bis 255 (Weiß). Die Bilder haben eine Abmessung von 100x100 Pixeln, deren Grauwerte in einer zweidimensionalen Reihung* `bild` *von Ganzzahlen gespeichert werden.*

a) *Implementieren Sie eine Operation* `hatFieber`*, die als Parameter einen Grauwert* `grenze` *erhält und die als Ergebnis einen Wahrheitswert liefert, der angibt, ob Pixel mit einer höheren Temperatur als der, die dem übergebenen Grauwert entspricht, im global definierten Bild enthalten sind.*

b) *Beschreiben Sie, wie die Operation* `hatFieber` *zu einer Operation* `swBild` *abgeändert werden kann, die das gegebene Bild in ein Schwarzweißbild verwandeln soll. Dazu wird der Grauwert* `grenze` *als Schwellwert benutzt. Alle Pixel mit einer höheren Temperatur als der Schwellwert werden weiß (255) gefärbt, die anderen schwarz (0).*

c) *Bereiche gleichen Grauwerts lassen sich durch Anwendung einer* `umfaerben`*-Operation mit einem anderen Grauwert färben, indem in einem zusammenhängenden Farbbereich von einem Punkt ausgehend alle umliegenden Punkte gleicher Farbe entsprechend verändert werden.*

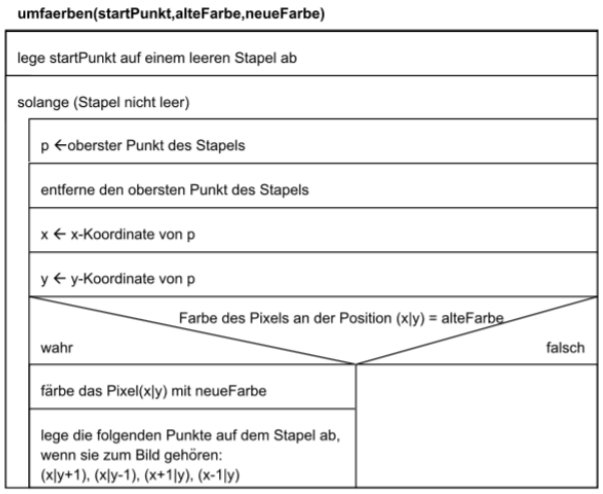


Abbildung 6.14.: Operation umfaerben

In der Abbildung ist das Struktogramm einer Operation `umfaerben` *gegeben, deren Parameter startPunkt die Bildkoordinaten des Punktes enthält, von dem aus der Umfärbeprozess im global vereinbarten Graubild* `bild` *starten soll. Implementieren Sie unter Benutzung der Klasse Stapel die Operation* `umfaerben`*. Definieren Sie dazu zuvor eine Klasse Punkt zur Verwaltung der x- und y-Koordinaten eines Punkts.*

d) Weil die Operation aus a) schon „Fieber“ feststellt, wenn ein einzelnes Pixel eine hohe Temperatur signalisiert, soll das Verfahren verbessert werden. Dazu wird das Bild wie in Aufgabenteil a) geschildert in ein Schwarzweißbild umgesetzt. In diesem werden weiße Bereiche gesucht. Größere weiße Bereiche sollen nach dem im Material in der Abbildung dargestellten Verfahren ermittelt werden.

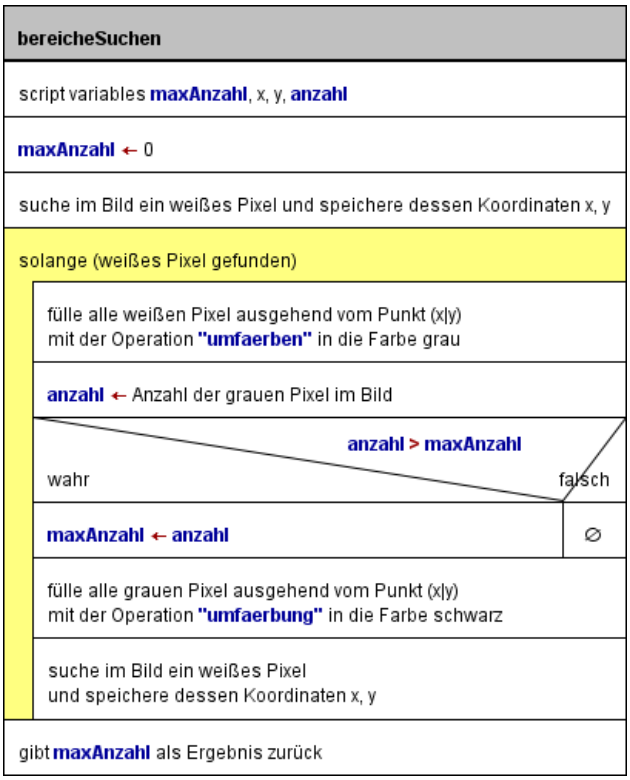


Abbildung 6.15.: Operation bereicheSuchen

Schätzen Sie begründet die Zeitkomplexität der Teilschritte der Operation `bereicheSuchen` *in Abhängigkeit von der Anzahl n der Pixel des Bildes ab.*

Untersuchen Sie die Zeitkomplexität der gesamten Operation `bereicheSuchen` *für die Fälle, dass das Bild kein weißes Pixel bzw. nur weiße Pixel enthält.*

Beschreiben Sie die Struktur eines Bildes, das eine sehr hohe Laufzeit von `bereicheSuchen` *bewirkt, und schätzen Sie auch für diesen Fall die Zeitkomplexität ab.*

Beurteilen Sie das Laufzeitverhalten der Operation `bereicheSuchen` *für reale Wärmebilder des oben geschilderten Szenarios.*

7. Bäume

In diesem Kapitel lernen Sie

- was ein Binärbaum und was ein binärer Suchbaum (binary search tree = BST) ist,
- welches Problem man mit Suchbäumen lösen kann,
- wie man Daten in einen binären Suchbaum einfügt,
- wie man einen Baum traversiert,
- wie man Binärbäume rekursiv durchläuft,
- wie man Binärbäume iterativ durchläuft.

7.1. Bäume und binäres Suchen

Ein **ungerichteter Baum** ist ein zusammenhängender kreisfreier ungerichteter Graph. Zusammenhängend bedeutet, dass der Graph nicht in mehrere Teile zerfällt, sondern dass alle Knoten miteinander verbunden sind. Kreisfrei heißt, dass es keine geschlossenen Kantenzüge gibt, bei denen Start- und Endpunkt zusammenfallen. Ungerichtet bedeutet, dass die alle Kanten keine Richtung haben. Der **Grad eines Knoten** v in einem ungerichteten Graphen ist die Anzahl der Nachbarn von v. Die Knoten mit Grad 1 heißen Blätter, die übrigen Knoten heißen innere Knoten.

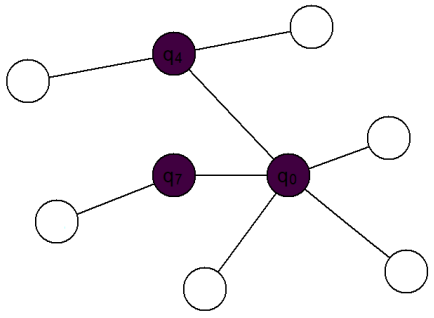
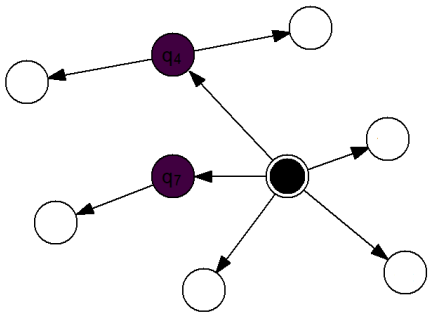


Abbildung 7.1.: Ungerichteter Baum

Ein **gerichteter** Baum ist ein gerichteter Graph, der ein ungerichteter Baum ist, wenn man die Richtungen der Kanten ignoriert. Er ist also ein gerichteter zusammenhängender kreisfreier Graph.

Ein **gewurzelter Baum** ist ein gerichteter von einem Knoten w aus zusammenhängender kreisfreier Graph. Der den Zusammenhang definierende Knoten w wird **Wurzel** genannt. Er hat den Eingangsgrad 0 und ist der einzige Knoten mit dieser Eigenschaft. Alle Knoten mit Ausgangsgrad 0 heißen **Blätter**. Alle Knoten mit positivem Ausgangsgrad heißen **innere Knoten**.



Schwarz mit Kreis: Wurzel; schwarz: innere Knoten; weiß: Blätter

Abbildung 7.2.: Gewurzelter Baum

Als **Binärbaum** bezeichnet man einen Graphen, der als gewurzelter Baum an jedem Knoten höchstens zwei Kindknoten besitzt. Meist wird verlangt, dass sich die Kindknoten eindeutig in linkes und rechtes Kind einteilen lassen. Ein Beispiel für einen solchen Binärbaum ist die Ahnentafel, bei der allerdings die Elternteile durch die Kindknoten zu modellieren sind.

Ein Binärbaum ist entweder leer oder er besteht aus einer Wurzel, einem linken und einem rechten Teilbaum, die wiederum Binärbäume sind. Meistens wird die Wurzel in graphischen Darstellungen oben und die Blätter unten platziert. Entsprechend ist ein Weg von der Wurzel zu den Blättern einer von oben nach unten.

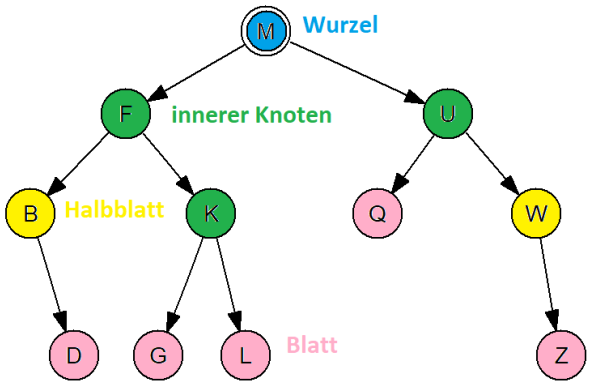


Abbildung 7.3.: Binärbaum

Knoten mit Ausgangsgrad ≥ 1 bezeichnet man als innere Knoten, solche mit Ausgangsgrad 0 als Blätter oder äußere Knoten. Bei Binärbäumen findet sich gelegentlich die Bezeichnung „Halbblatt“ für einen Knoten mit Ausgangsgrad 1 (englisch *non-branching node*)

Ein **binärer Suchbaum**, häufig abgekürzt BST (Binary Search Tree), ist ein binärer Baum, bei dem die Knoten „Schlüssel“ tragen, und die Schlüssel des linken Teilbaums eines Knoten nur kleiner (oder gleich) und die des rechten Teilbaums nur größer (oder gleich) als der Schlüssel des Knotens selbst sind.

Suchbäume sind Lösungen des sog. „Wörterbuch-Problems“. Angenommen, es gibt eine große Anzahl von Schlüsseln, denen jeweils ein Wert beigegeben sind. In einem Wörterbuch deutsch-englisch ist das deutsche Wort der Schlüssel und englische Wörter sind der Wert. Ähnlich ist es im Telefonbuch: Name und Adresse sind Schlüssel, die Telefonnummer der Wert.

In beiden Beispielen sind üblicherweise die Schlüssel sortiert. Das erlaubt es, das Buch in der Mitte aufzuschlagen und zu überprüfen, ob der Suchbegriff vor oder nach dem Schlüssel liegt. Der Rest des Buches ist wieder ein zusammenhängendes Segment, das wieder halbiert wird, usw., bis der Suchbegriff gefunden wurde.

Diese Vorgehensweise hat in der Informatik den Namen **binäres Suchen**. Dagegen wäre **sequentielles Suchen** das Suchen in einer Liste. Man geht von vorn beginnend die Liste durch, bis man den gesuchten Eintrag gefunden hat.

Anders als beim binären Suchen muss beim sequentiellen Suchen der Suchbegriff potentiell mit allen Schlüsseln verglichen werden (Dafür braucht die Eingabe nicht sortiert zu sein).

Der Unterschied zwischen beiden Verfahren kann erheblich sein. In einem Telefonbuch mit $n = 2^{20} - 1 = 1.048.575$ Einträgen müssen beim sequentiellen Suchen im Mittel $(1048575 + 1)/2 = 524.288$ Vergleiche durchgeführt werden. Beim binären Suchen gelangt man nach maximal $log_2(2^{20} - 1 + 1) = 20$ Vergleichen zum selben Ergebnis.

Diese Formel kann man sich wie folgt klarmachen: Ein vollständig ausgefüllter Baum mit zwei Ebenen hat $2^2 - 1 = 3$ Elemente. Hier reichen zwei Vergleiche, um ein gesuchtes Element zu finden. Fügen wir eine dritte Ebene hinzu, so hat der Baum $2^3 - 1 = 7$ Elemente. Jetzt reichen drei Vergleiche. Bei der vierten Ebene kommen acht Elemente hinzu, damit hat der Baum $2^4 - 1 = 15$ Elemente. Es reichen vier Vergleiche. Bei einem Baum mit $2^n - 1$ Elementen reichen n Elemente. Für unseren Baum mit $2^{20} - 1 = 1.048.575$ Einträgen reichen also 20 Vergleiche, um einen Eintrag zu finden.

7.2. Implementierung des Binärbaums

Auch der Binärbaum gehört zu den Objekten, die im Zentralabitur vorgegeben sind. Der Binärbaum ist ein schönes Beispiel für die Leistungsfähigkeit von Snap!. Fast alle Operationen lassen sich als Einzeiler programmieren, wenn wir den kleinen Trick aus 2.3 anwenden (statt `IF ... ELSE ...` eine Aussage reporten).

Ausgestaltung und Anzahl der Operationen von Binärbäumen im Abitur schwanken sehr stark. In der aktuellen Fassung gibt es insgesamt 15 Operationen. Dazu gehören zwei Konstruktoren, einen für einen leeren Baum und einen zweiten für einen Baum mit Inhalt. Eine Besonderheit stellt das Predicate `isLeaf` dar. Die übrigen Operationen zerfallen in drei Gruppen. Es gibt jeweils eine has-, get-, set- und delete-Operation für die Wurzel (Item), den linken und den rechten Teilbaum.

Wir implementieren den Baum mit Hilfe von

Binärbaum
-Dynamische Reihung aus Elementen vom Inhaltsty Die Nummerierung beginnt mit Null
c BinTree() c BinTree(inhalt:Inhaltstyp) +hasItem(): Wahrheitswert +getItem(): Inhaltstyp +setItem(inhalt:Inhaltstyp) +deleteItem() +isLeaf(): Wahrheitswert +hasLeft(): Wahrheitswert +getLeft(): Binärbaum +setLeft(b:Binärbaum) +deleteLeft() +hasRight(): Wahrheitswert +getRight(): Binärbaum +setRight(,b:Binärbaum) +deleteRight()

Abbildung 7.4.: Objekt Binärbaum

verketteten Listen. Jeder Knoten besteht dabei aus einer dreielementigen Liste. Das erste Element enthält den Inhalt der Wurzel, 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. Felder ohne Wert kennzeichnen wir mit **null**.

Der Konstruktor `binTree()` liefert eine dreiteilige Liste. Dabei schreiben wir in alle Elemente **null**.

Abbildung 7.5.: Konstruktor für einen leeren Baum

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

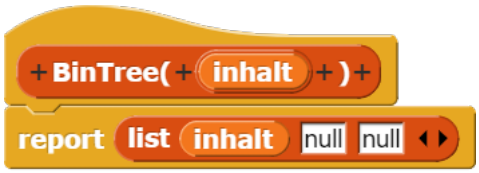


Abbildung 7.6.: Konstruktor für einen Baum mit Inhalt

Das Predicate `isLeaf` prüft, ob der aktuelle Knoten ein Blatt ist, d.h., ob er keine Teilbäume hat.

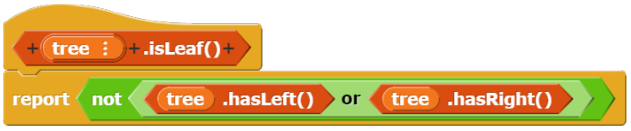


Abbildung 7.7.: binTree.isLeaf()

Das Predicate `hasItem()` prüft, ob der aktuelle Knoten einen Inhalt hat.

Abbildung 7.8.: binTree.hasItem()

Der Reporter `getItem()` liefert den Inhalt der Wurzel.

Abbildung 7.9.: binTree.getItem()

Das Command `setItem(inhalt)` überschreibt den Inhalt der aktuellen Wurzel mit dem neuen Inhalt.

Abbildung 7.10.: binTree.setItem(inhalt)

Das Command `deleteItem()` löscht den Inhalt der Wurzel. Dabei entsteht ein Problem: Wird die Wurzel eines Knotens gelöscht, der noch Teilbäume hat, dann ist der Baum kein Binärbaum mehr. Damit gehen alle Vorteile, die mit der Datenstruktur Binärbaum verbunden sind, verloren. Eine solche Löschoperation findet sich in der Fachliteratur nicht[1]. Es gibt

[1] z.B. H. Herold, B. Lurz, J. Wohlrab, Grundlagen der Informatik, München: Pearson 2007, S. 300ff.; H. Balzert, Lehrbuch Grundlagen der Informatik, München: Spektrum 2005, S. 619ff.; D. Knuth, The

aber Operationen zum Entfernen eines Knoten[2]. Wir schränken deshalb die Funktionsweise dieses Blocks in der Weise ein, dass wir das Löschen nur durchführen, wenn weder ein linker noch ein rechter Teilbaum vorhanden sind. Gibt es noch Teilbäume, dann geben wir eine Fehlermeldung aus.

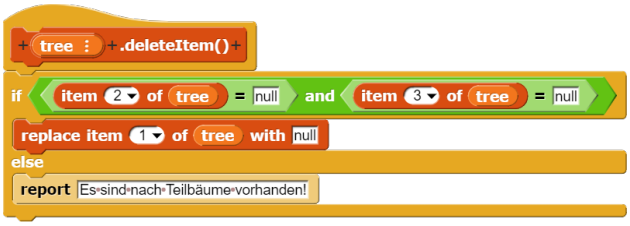


Abbildung 7.11.: binTree.deleteItem()

Die Operationen für die Teilbäume lasse sich entsprechend implementieren. Nur beziehen sich die Operationen für den linken Teilbaum auf das zweite Element der Liste und die Operationen für den rechten Teilbaum auf das dritte Element der Liste.

Abbildung 7.12.: binTree.hasLeft()

Der Reporter `getLeft()` liefert den linken Teilbaum.

Abbildung 7.13.: binTree.getLeft()

Das Command `setLeft(b: Binärbaum)` überschreibt den linken Teilbaum des aktuellen Baums mit dem als Parameter übergebenen Binärbaum.

Die Löschoperationen für die Teilbäume sind unproblematisch.

Abbildung 7.14.: binTree.setLeft(inhalt)

Abbildung 7.15.: binTree.deleteLeft()

7.3. Einfügen in Binärbäume

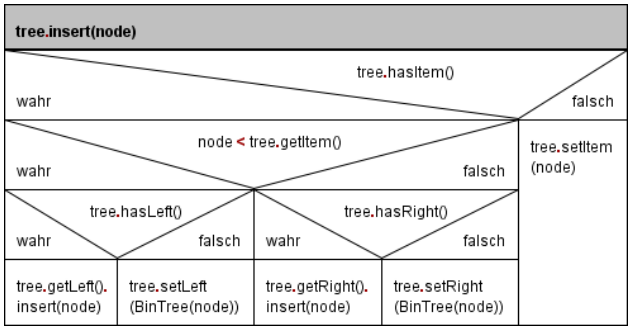


Abbildung 7.16.: Einfügen in einen Binärbaum

Beim Einfügen eines Elements in einen binären Suchbaum verwenden wir ein dreistufiges Verfahren. Im ersten Schritt prüfen wir, ob der Baum eventuell leer ist. Falls das der Fall ist, fügen wir das Element in die Wurzel ein und sind fertig. Wenn der Baum nicht leer ist, prüfen wir, ob das einzufügende Element kleiner ist als die Wurzel. Je nach dem Ergebnis der Prüfung untersuchen wir die linke oder die rechte Seite.

Ist das Element kleiner als die Wurzel und es gibt einen linken Teilbaum, dann rufen wir die Operation zum Einfügen rekursiv mit dem linken Teilbaum auf. Ist das Element kleiner als die Wurzel und es gibt keinen linken Teilbaum, dann erzeugen wir einen neuen Binärbaum mit dem Element als Wurzel und fügen diesen als linken Teilbaum in unseren Baum ein.

Art of Computer Programming, Vol. I Fundamental Algorithms, Boston [3]1997, S. 308ff.

[2] z.B. Balzert, ebd., S. 621

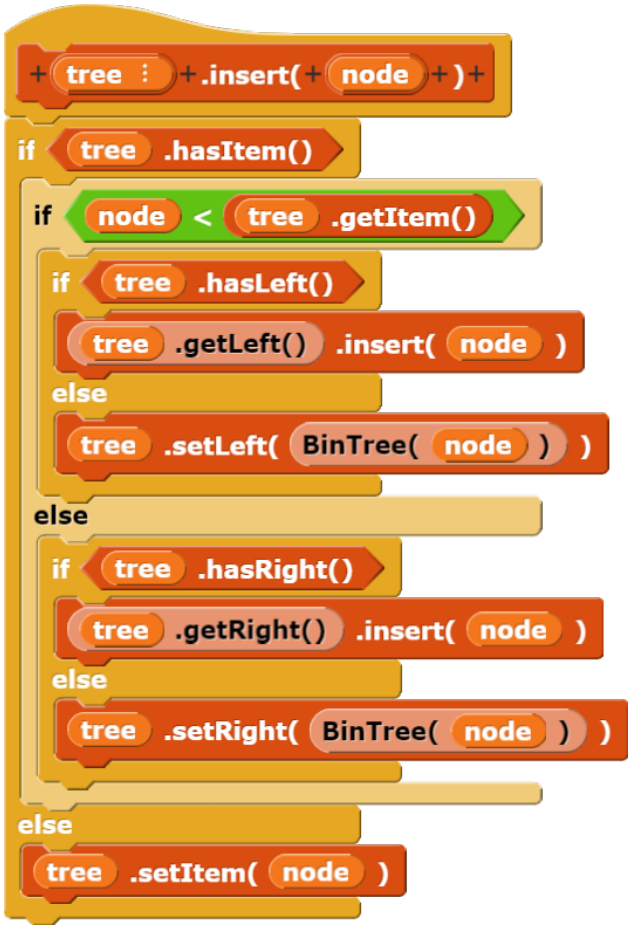


Abbildung 7.17.: Einfügen in einen Binärbaum

Ist das Element größer als die Wurzel und es gibt einen rechten Teilbaum, dann rufen wir die Operation zum Einfügen rekursiv mit dem rechten Teilbaum auf. Ist das Element größer als die Wurzel und es gibt keinen rechten Teilbaum, dann erzeugen wir einen neuen Binärbaum mit dem Element als Wurzel und fügen diesen als rechten Teilbaum in unseren Baum ein.

Die Abbruchbedingung für den rekursiven Aufruf ist, wenn es keinen linken bzw. rechten Teilbaum mehr gibt. Dann wird der Knoten eingefügt.

7.4. Rekursive Methode für Binärbäume

Eine rekursive Methode für Binärbäume setzt sich aus vier Teilen zusammen:

- dem Auslesen der Wurzel (auch „Wurzelbehandlung“ genannt).
- den rekursiven Aufrufen der Methode. Meist benötigt man zwei rekursive Aufrufe, einen für den linken und einen für den rechten Teilbaum. Bei Methoden, die etwas zurückgeben, muss man den Rückgabewert verarbeiten.

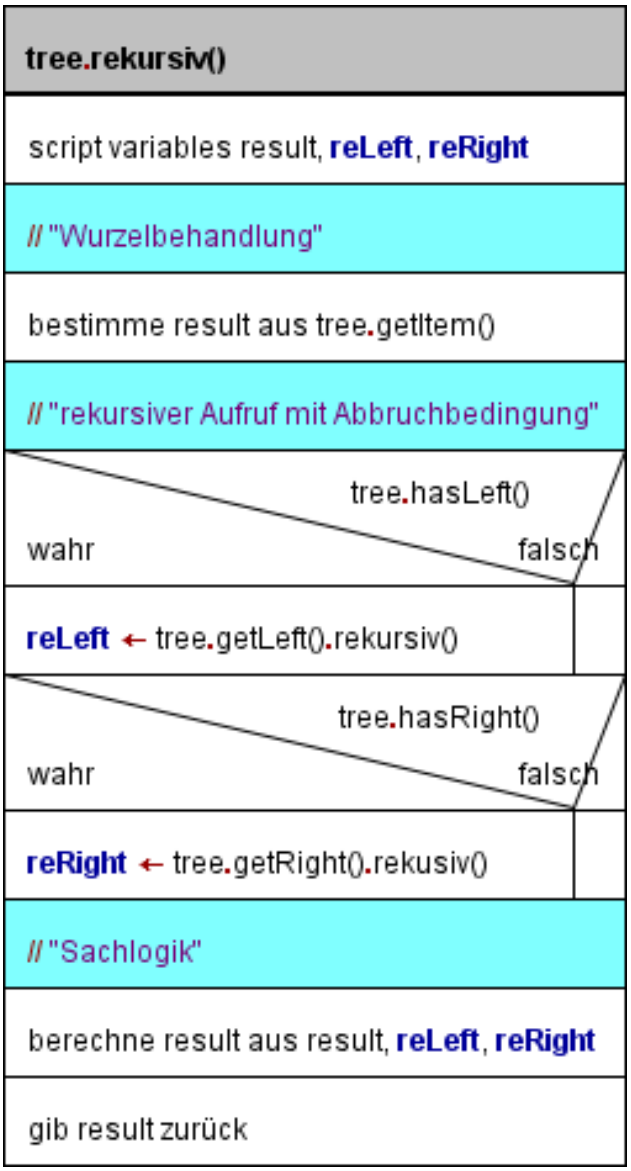


Abbildung 7.18.: Rekursive Methode für Binärbäume

- eine Abbruchbedingung oder mehrere Abbruchbedingungen. Bei Bäumen wird als Abbruchbedingung die Abfrage `tree.hasLeft()` bzw. `tree.hasRight()` verwendet. Der rekursive Aufruf der Methode erfolgt nur dann, wenn es einen linken bzw. rechten Teilbaum gibt.
- eine „Sachlogik“. In der Sachlogik muss

man die Frage beantworten, wie sich das Gesamtergebnis aus der Wurzel und aus den Ergebnissen des linken bzw. rechten Teilbaums zusammensetzt.

7.5. Traversierung

Traversierung bezeichnet das systematische Untersuchen der Knoten des Baumes in einer bestimmten Reihenfolge. Es gibt verschiedene Möglichkeiten, die Knoten von Binärbäumen zu durchlaufen. Man unterscheidet die folgenden Varianten (dabei ist „Durchlaufen der Teilbäume“ L und R rekursiv gemeint):

- pre-order Wurzel - Linker Baum - Rechter Baum
- in-order Linker Baum - Wurzel - Rechter Baum
- post-order Linker Baum - Rechter Baum -Wurzel

Dabei wird immer der linke Teilbaum vor dem rechten Teilbaum durchlaufen. Die Vorsilbe (pre/in/post) gibt an, an welcher Stelle die Wurzel W durchlaufen wird.

Traversierung in pre-order, in-order und post-order Reihenfolge ist eine rekursive Methode zur Ausgabe eines Binärbaums. Die Wurzel wird jeweils als Element in eine Liste eingelesen („Wurzelbehandlung“). Die rekursiven Aufrufe erfolgen mit dem linken bzw. rechten Teilbaum. Die Abbruchbedingung wird erfüllt, wenn Teilbäume nicht vorhanden sind. Die Sachlogik schließlich setzt die Listen mit den Teilergebnissen in der gewünschten Reihenfolge zusammen.

pre-order oder Hauptreihenfolge

W-L-R: Zuerst wird die Wurzel W betrachtet und anschließend der linke, dann der rechte Teilbaum.

Die Methode packt zunächst die Wurzel in eine Liste, weil `append` Listen als Parameter verlangt. Dann wird das Ergebnis des linken Teilbaums angefügt, falls ein solcher vorhanden ist, und dann das Ergebnis des rechten Teilbaums.

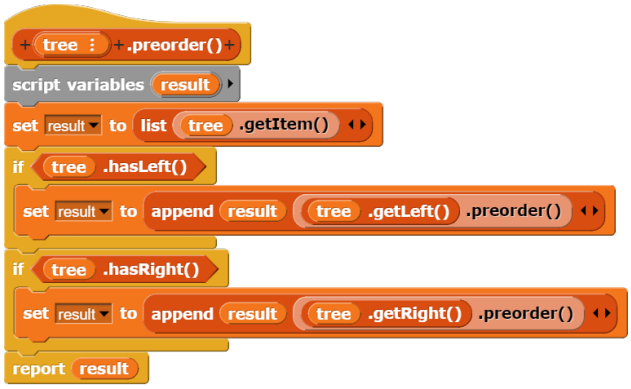


Abbildung 7.19.: Pre-Order

Wird ein binärer Suchbaum in pre-order-Traversierung ausgegeben, dann kann die entstehende Liste benutzt werden, um den Baum wieder zu rekonstruieren.

in-order oder symmetrische Reihenfolge

L-W-R: Zuerst wird der linke Teilbaum durchlaufen, dann die Wurzel W und anschließend der rechte Teilbaum.

Auch hier packt die Methode den Inhalt der Wurzel in eine Liste. Das Ergebnis des linken Teilbaums wird dann links angefügt, das Ergebnis des rechten Teilbaums rechts.

Die in-order-Traversierung liefert bei binären Suchbäumen die Anordnung der Schlüssel nach der Größe.

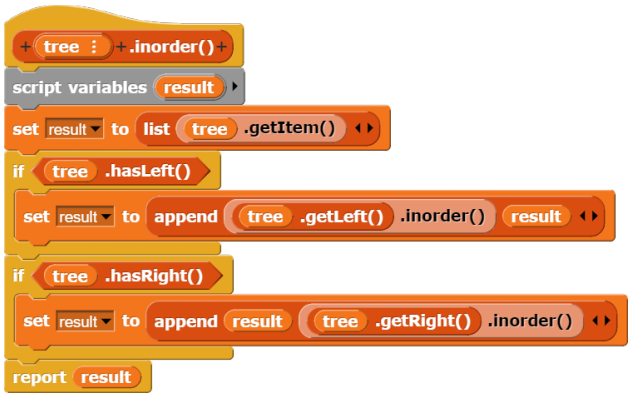


Abbildung 7.20.: In-Order

post-order oder Nebenreihenfolge

L-R-W: Zuerst wird der linke Teilbaum durchlaufen, dann der rechte Teilbaum und anschließend die Wurzel W.

level-order

level-order bedeutet das Anordnung der Knoten gewissermaßen in Schichten von links nach rechts und von oben nach unten. Im Unter-

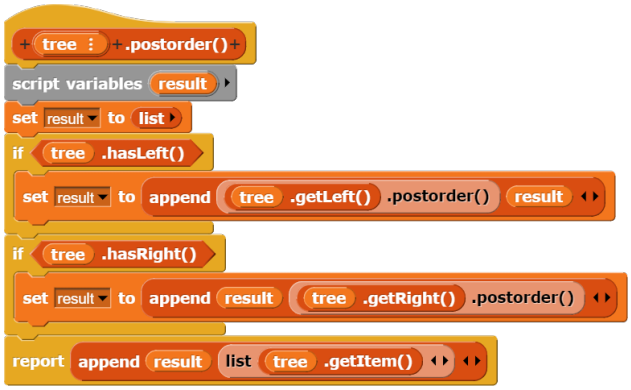


Abbildung 7.21.: Post-Order

schied zu den drei anderen Traversierungsverfahren kann man level-order am einfachsten programmieren, wenn man iterativ durch einen Binärbaum geht.

Eine zeichnerisches Lösungsverfahren für Traversierungsaufgaben findet sich bei Youtube unter https://www.youtube.com/watch?v=5X8CkFBq_8k.

7.6. Suchen in Binärbäumen

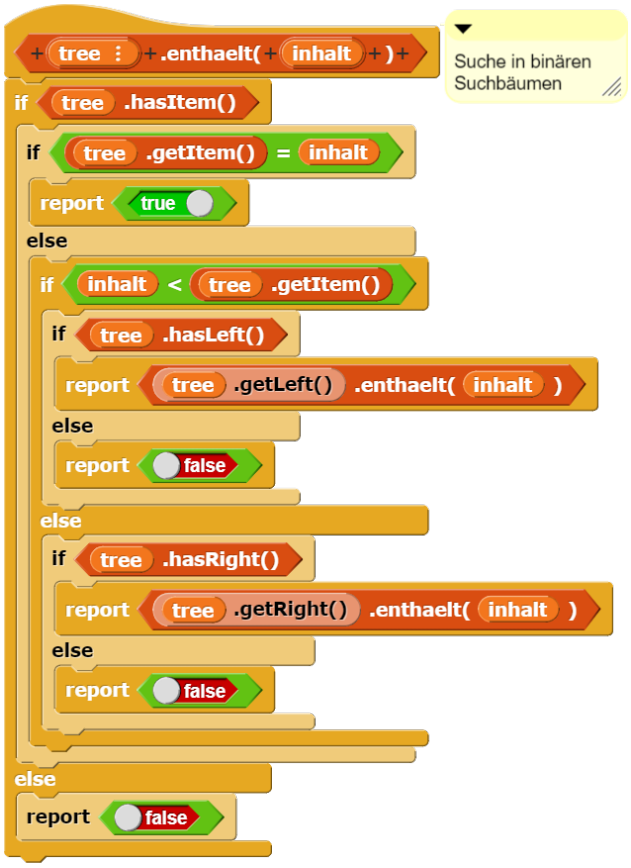


Abbildung 7.22.: Suchen im Baum

Auch das Suchen in Binärbäumen kann als rekursive Methode implementiert werden. Zunächst wird überprüft, ob der Baum leer ist. Falls das zutrifft, ist das gesuchte Element nicht enthalten und die Methode gibt `false` zurück. Im zweiten Schritt muss dann überprüft werden, ob die Wurzel das gesuchte Element enthält. Falls ja, gibt die Methode `true` zurück. Im dritten Schritt wird überprüft, ob das gesuchte Element kleiner ist als der aktuelle Knoten. Trifft dies zu, so wird im vierten Schritt geprüft, ob es einen linken Teilbaum gibt. Fehlt dieser, dann gibt die Methode `false` zurück. Gibt es einen linken Teilbaum, dann erfolgt der rekursive Aufruf der Methode mit dem linken Teilbaum. Ist das gesuchte Element nicht kleiner als der aktuelle Knoten, so wird im vierten Schritt geprüft, ob es einen rechten Teilbaum gibt. Fehlt dieser, dann gibt die Methode `false` zurück. Gibt es einen rechten Teilbaum, dann erfolgt der rekursive Aufruf der Methode mit dem rechten Teilbaum.

Auch diese Methode entspricht unserem Schema. Bei der Wurzelbehandlung wird überprüft, ob die Wurzel leer ist und ob die Wurzel das gesuchte Element bereits enthält. Der rekursive Aufruf erfolgt in Abhängigkeit von der Größe des gesuchten Elements und der Existenz des entsprechenden Teilbaum. Die Abbruchbedingungen sind die Fälle, in denen ein Ergebnis eindeutig festgestellt werden kann bzw. wenn keine weiteren Teilbäume vorliegen. Die Sachlogik erfolgt über die Rückgabe des Wahrheitswertes der beiden Teilbäume, wobei auf jeder Stufe nur ein Teilbaum ausgewertet wird.

7.7. Weitere rekursive Operationen

Die `Anzahl der Knoten` eines Binärbaumes kann ebenfalls rekursiv erfolgen. Zur Wurzelbehandlung gehört, dass jeder Teilbaum einen Knoten als Wurzel hat und deshalb Eins zum Ergebnis hinzufügt, wenn dieser Knoten nicht leer ist. Die Anzahl der Knoten in den beiden Teilbäumen wird rekursiv bestimmt. In der Sachlogik müssen Eins, das Ergebnis des

linken Teilbaums und das Ergebnis des rechten Teilbaums addiert werden.

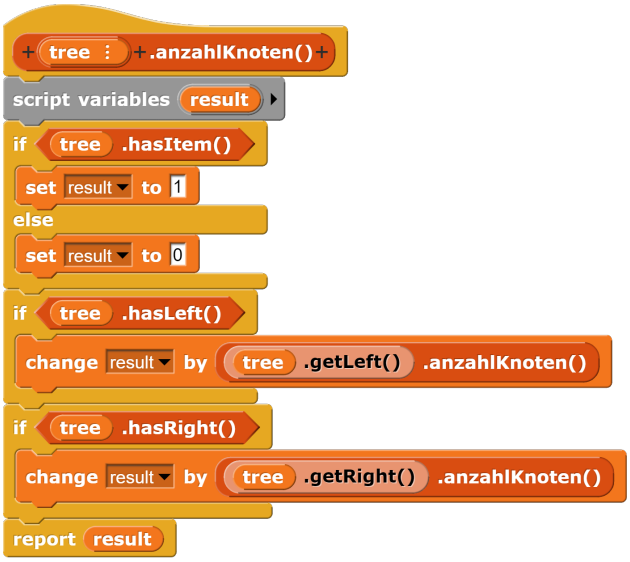


Abbildung 7.23.: Anzahl der Knoten

Die `Anzahl der Blätter` erhält man ähnlich: Ist der aktuelle Knoten ein Blatt, dann trägt er Eins zum Gesamtergebnis bei. Ist der aktuelle Knoten kein Blatt, dann müssen die Ergebnisse des linken und des rechten Teilbaums addiert werden.

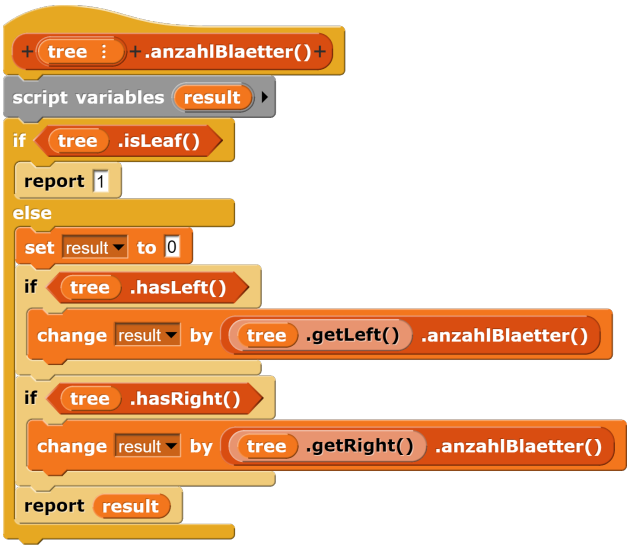


Abbildung 7.24.: Anzahl der Blätter

Gesucht ist eine `Liste aller Blätter` eines Binärbaumes. Tree.isLeaf() findet die Blätter. `Result` kann in Abhängigkeit von der Wurzel auf leere Liste oder Liste, die die Wurzel enthält, gesetzt werden. In der Sachlogik müssen die Teillisten mit den Blättern zusammengefügt werden.

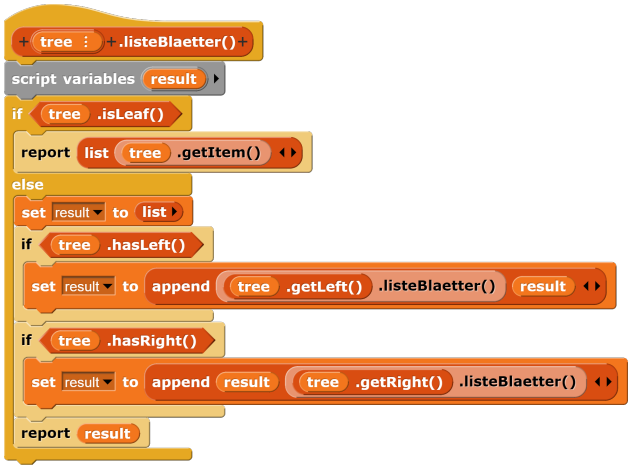


Abbildung 7.25.: Liste der Blätter

7.8. Linearisierung von Binärbäumen

Eine Alternative zum rekursiven Durchgehen durch Bäume ist das iterative Durchgehen. Dazu **linearisieren** wir den Baum, d.h. wir packen die Knoten in eine Liste, die wir dann auswerten können. Das klassische Beispiel für die Linearisierung von Bäume ist das Traversierungsverfahren **levelorder**.

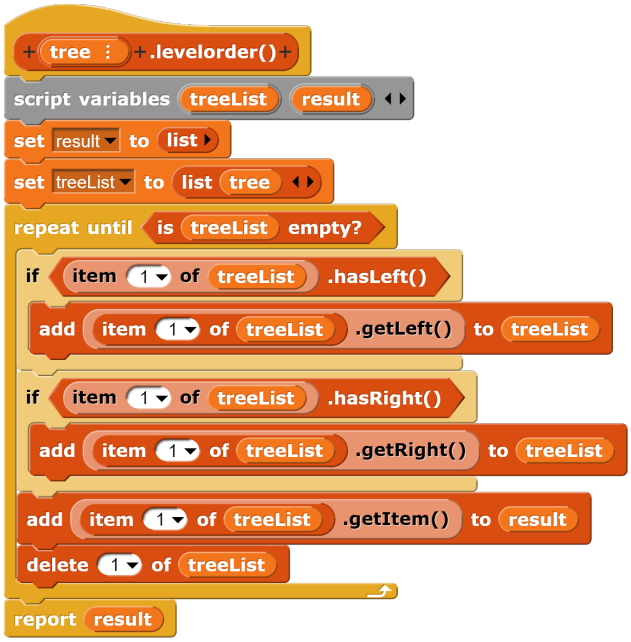


Abbildung 7.26.: Levelorder

Als Skriptvariablen benötigen wir außer dem Result eine Liste zur Aufnahme der Teilbäume, die wir `treeList` nennen. Wir initialisieren `result` als leere Liste und `treeList` als Liste, die den Baum als Element enthält. Dann

wiederholen wir die folgenden Schritte, bis `treeList` leer ist:

- Wenn das erste Element von `treeList` einen linken Teilbaum enthält, fügen wir diesen hinten an `treeList` an.
- Wenn das erste Element von `treeList` einen rechten Teilbaum enthält, fügen wir diesen hinten an `treeList` an. Damit haben wir die beiden Teilbäume ausgelesen.
- Jetzt erfolgt die Sachlogik. In diesem Fall wird die Wurzel des ersten Elements von `treeList` zu `result` hinzugefügt, weil wir die Knoten in der Reihenfolge levelorder, d.h. von links nach rechts und von oben nach unten, ausgeben wollen.
- Das erste Element von `treeList` wird gelöscht und wir fahren mit dem nächsten fort.

Viele Aufgaben mit Binärbäumen lassen sich wahlweise rekursiv oder durch Linearisierung lösen. So kann unser levelorder-Block ganz einfach zu einem Block `listeBlaetter` umgebaut werden, indem wir in der Sachlogik abfragen, ob es sich um ein Blatt handelt und nur in diesem Fall das Element in die Ergebnisliste aufnehmen.

7.9. Termbäume

Vorbemerkung: Ein Termbaum ist kein binärer Suchbaum, sondern ein Binärbäum, der dazu benutzt werden kann, arithmetische Ausdrücke zu berechnen.

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 sind zwei Operationen, die den zugehörigen mathematischen Term mit Klammern a) berechnen und b) ausgeben. 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 der hier

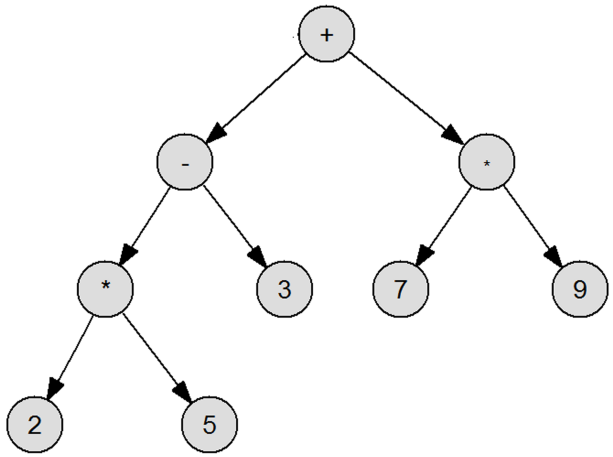


Abbildung 7.27.: Termbaum

betrachteten Rechenoperation Addition, Subtraktion, Multiplikation und Division zwei Zahlen gehören.

Falls der aktuelle Knoten eine Zahl enthält, können wir diese zurückgeben. Falls er ein Rechenzeichen enthält, müssen wir das Ergebnis berechnen in Abhängigkeit von den beiden Ergebnissen der Teilbäume und dem Rechenzeichen.

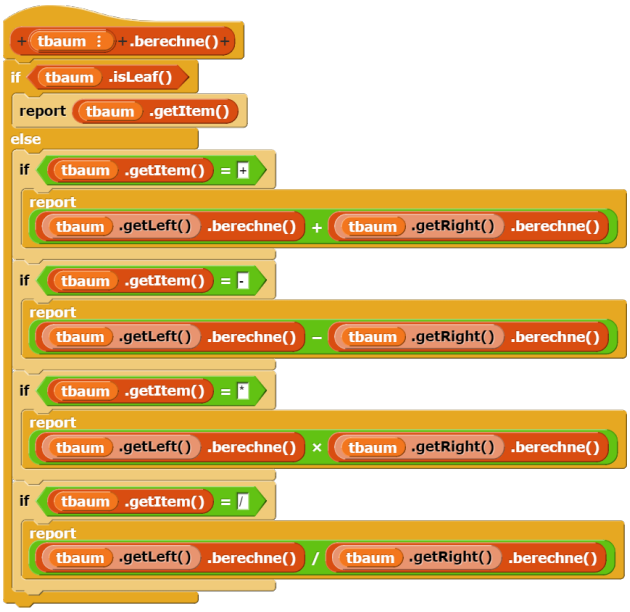


Abbildung 7.28.: Termbaum berechnen

Die Ausgabe eines Termbaums gestaltet sich noch einfacher:

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.

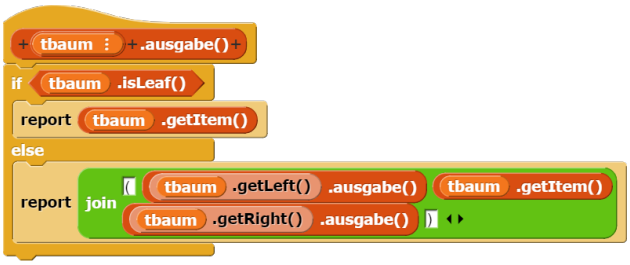


Abbildung 7.29.: Ausgabe des Termbaums

7.10. Aufgaben

Aufgabe 7.1 *Implementieren Sie die in dem UML-Diagramm in 7.2 beschriebenen Operationen für den Binärbaum.*

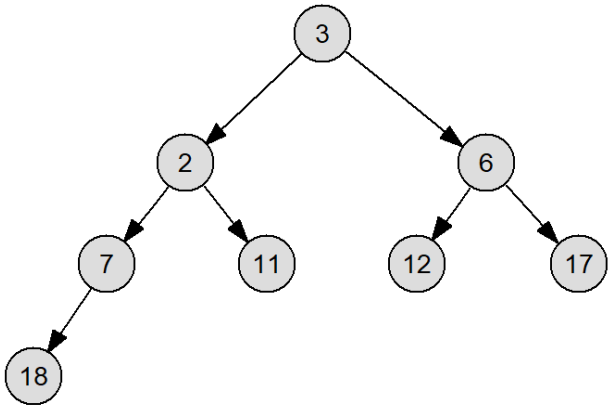


Abbildung 7.30.: Baum zu Aufgabe 7.2

Aufgabe 7.2 *Begründen Sie, dass es sich in Abbildung 7.27 zwar um einen Binärbaum, aber nicht um einen binären Suchbaum handelt.*

Geben Sie die Reihenfolge der besuchten Knoten nach Pre-, In-, Post- und Levelorder an!

Aufgabe 7.3 *Implementieren Sie eine Operation zum Einfügung in einen binären Suchbaum.*

Aufgabe 7.4 *Implementieren Sie eine Operation zur pre-order-, in-order- und zur post-order-Traversierung eines Binärbaums. Ergänzen Sie eine Operation zur level-order-Traversierung.*

Aufgabe 7.5 *Implementieren Sie eine Operation zum Suchen in einem binären Suchbaum.*

Aufgabe 7.6 *Implementieren Sie je eine Operation, die*

- *die Anzahl der Blätter*
- *die Anzahl linker Söhne*
- *die Höhe (Die Höhe eines Binärbaums ist definiert als die Anzahl der Knoten auf dem längsten Weg von der Wurzel zu einem Blatt)*

eines beliebigen binären Baumes bestimmt.

Aufgabe 7.7 *Gegeben sei ein beliebiger binärer Baum (siehe Beispiele oben). Jeder Knoten beinhalte eine Ganzzahl (Integer). Implementieren Sie eine Operation, die testet, ob sich ein Knoten mit einer bestimmten Ganzzahl (Integer) im Baum befindet!*

Aufgabe 7.8 *Implementieren Sie eine Operation, die testet, ob ein gegebener Binärbaum ein Suchbaum ist.*

Aufgabe 7.9 *Implementieren Sie eine Operation, die eine aufsteigend sortierte Liste so in einen binären Suchbaum einfügt, dass der Baum nach dem Einfügen aller Elemente balanciert ist.*

Hinweis: Die Balance eines Baumes ist definiert als (Höhe des rechten Teilbaums) - (Höhe des linken Teilbaums). Ein Baum ist balanciert, wenn der Betrag der Balance für jeden Knoten höchstens 1 beträgt.

Aufgabe 7.10 *Implementieren Sie eine iterative Operation zur Inorder-Traversierung eines binären Baumes. Jeder Knoten enthalte eine Ganzzahl. Zur Zwischenspeicherung der Knoten steht ein Stack zur Verfügung.*

8. Häufige Fehler

8.1. Automaten und Grammatiken

8.1.1. Fehler beim Zählen

Ein häufiger Fehler besteht darin, dass man übersieht, dass ein Automat nur zählen kann, indem er in einen neuen Zustand geht. Eine Schleife an einem Zustand bedeutet, dass man hier beliebig viele Wiederholungen einbauen kann. Damit ist ein exaktes Zählen ausgeschlossen.

Das gilt besonders für Automaten, die mit einem bestimmten Muster beginnen. Hier darf es während dieses Musters weder Schleifen noch Rückschleifen auf die ersten Zustände geben, weil man sonst nicht mehr unterscheiden kann, ob es der 1. bzw. 2. Buchstabe ist oder ein späterer.

Entsprechendes gilt für Automaten, die mit einem bestimmten Muster enden: Hier kann es am Ende keine Trap geben, weil das Muster ja noch später kommen kann.

Weitere Fehler entstehen dadurch, dass man nicht weit genug zählt. Wenn ein Automat ein bestimmtes Muster genau zwei Mal enthalten soll, muss man bis Drei zählen, um in eine Trap zu gelangen. Erst dann kann man sicher sein, dass das Muster zu oft enthalten ist.

8.1.2. Ausgabe und Zustand verwechselt

Der häufigste Fehler bei Mealy-Automaten ist, dass man Ausgaben und Zustände verwechselt. Betrachten wir einen Kaffeeautomaten: Er soll Kaffee ausschenken. Kaffee ist hier also eine Ausgabe, kein Zustand.

8.1.3. Akzeptierende Zustände bei Mealyautomaten

Ein Mealy-Automat hat keine akzeptierenden Zustände.

8.1.4. Ausgabe bei DEAs

Ein DEA oder NEA hat keine Ausgabe. Er meldet nur, ob ein Wort zu einer durch den Automaten definierten Sprache gehört oder nicht.

8.1.5. Grammatik ohne terminierende Regel

Jede Grammatik benötigt mindestens eine terminierende Regel, d.h. einer Regel der Form $A \rightarrow a$, in Worten: ein Nichtterminalsymbol wird durch eine Terminalsymbol ersetzt. Das liegt daran, dass die Worte jeder Sprache nur aus Buchstaben bestehen und keine Nichtterminalsymbole enthalten dürfen. Fehlt also eine terminierende Regel, dann kann kein Wort der Sprache gebildet werden.

8.1.6. Alle Worte beginnen mit dem Startsymbol

Alle Worte einer Sprache, die durch eine Grammatik beschrieben werden, beginnen mit dem Startsymbol, nicht mit einem beliebigen Symbol.

8.2. Huffman-Codierung

Ein häufiger Fehler in der Huffman-Kodierung ist, dass man zwar anfangs die Knoten nach ihrer Häufigkeit anordnet, bei der weiteren Konstruktion des Baumes aber dann nebeneinander liegende Knoten zusammenfasst und nicht auf jeder Stufe die zwei mit dem geringsten Gewicht.

8.3. Programmieren

8.3.1. Wertzuweisungen

Der häufigste Fehler ist die Verwendung falscher Befehle bei der Wertzuweisung. Dabei muss unterschieden werden, ob eine Variable für sich steht oder Element einer Liste ist:

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

Abbildung 8.1.: Set und Change

Ein Befehl wie `set item i of myList to 10` ist falsch. Der `SET`-Befehl lässt sich nicht auf Elemente von Listen anwenden. In Snap! ist das Feld nach dem `SET` nur durch den kleinen schwarzen Pfeil nach unten einstellbar, d.h. es dürfen als Ziel nur Variable (einschließlich Parameter und Script Variables) ausgewählt werden.

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 8.2.: Replace

Die Verwendung eines falschen Blocks kann 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.

Entsprechendes gilt für Listen. Hat man eine Liste durch `set myList to list{A, B, C}` erzeugt und wendet dann einen `SET`-Befehl an, wird die Liste gelöscht und `myList` erhält den Wert, der im zweiten SET-Befehl zugewiesen wird.

8.3.2. Zählschleifen

Bei der händischen Implementierung von Zählschleifen gibt es viele Fehlermöglichkeiten. Deshalb wird dringend empfohlen, `FOR`-Schleifen zu verwenden.

Einer der häufigsten Fehler bei geschachtelten Zählschleifen ist, dass an der falschen Stelle initialisiert wird. Beide Schleifen müssen jeweils unmittelbar vor der Wiederholungsanweisung initialisiert werden.

8.3.3. 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 hoch zählt. 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`.

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`.

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,

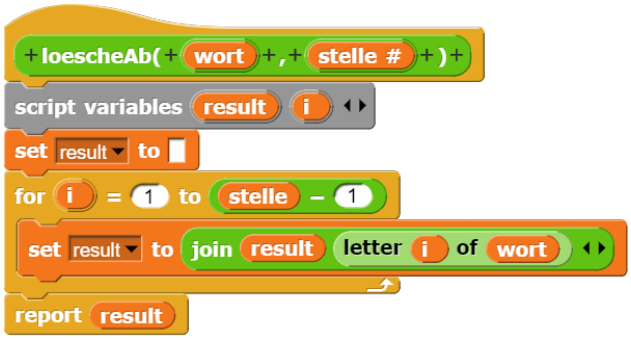


Abbildung 8.3.: loescheAb(wort, stelle) mit Fehler

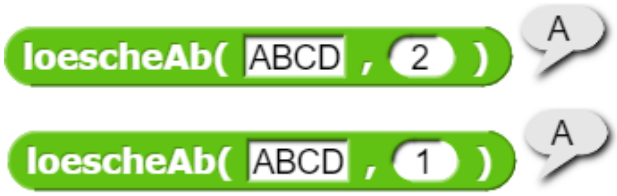


Abbildung 8.4.: Result von loescheAb

0. Der erste Buchstabe wird an `result` angehängt, einen nullten Buchstaben gibt es nicht. Deshalb wird `A` ausgegeben

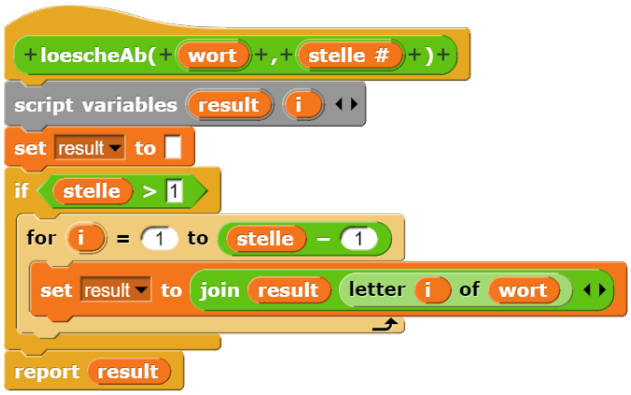


Abbildung 8.5.: loescheAb korrigiert

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.

8.3.4. Wiederholungen

In den meisten Programmiersprachen ist die `REPEAT`-Schleife eine nachprüfende Schleife mit einer Abbruchbedingung. In Snap! ist die `REPEAT UNTIL`-Schleife eine vorprüfende Schleife mit einer Abbruchbedingung. Falls eine Wiederholungsbedingung vorliegt, muss diese verneint werden.

Wird die `REPEAT UNTIL`-Schleife als Zählschleife verwendet, dann muss die Grenze entsprechend angepasst werden. Bei `REPEAT UNTIL i=Endwert` wird die Schleife für den Endwert nicht mehr ausgeführt.

8.3.5. 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 einer einfach verketteten Liste 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!

Auf eine Schlange darf also nur mit den Operationen `Schlange.Queue()`, `Schlange.isEmpty()`, `Schlange.enqueue (inhalt)`, `Schlange.dequeue()` und `Schlange.head()` zugegriffen werden, **nie** mit `item i of Schlange` oder `length of Schlange`! Entsprechendes gilt für einen Stapel. Hier sind nur `Stapel.Stack()`, `Stapel.isEmpty()`, `Stapel.pop()`, `Stapel.top()` und `Stapel.push()` zulässig.

8.3.6. 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 oder zwei Listen zu verbinden! Leider legt der Name des Blocks eine solche Fehlinterpretation nahe.

8.3.7. Parameter

Parameter werden beim Aufruf eines Blocks übergeben. Bei selbstgeschriebenen Blöcken stehen sie im Quelltext in Klammern in der Kopfzeile, wobei der Typ in der Kopfzeile **und nur in der Kopfzeile** hinter einem Doppelpunkt angegeben wird (Beispiel: `codeCaesar( text: Zeichenkette, key: Buchstabe): Zeichenkette`. Bei Snap!-Blöcken kann auf die Klammern verzichtet werden, z.B. `move 10 steps`. Es ist weder notwendig noch sinnvoll, den Wert von Parametern durch eine Benutzerabfrage oder eine Zufallsfunktion festzulegen. Damit zerstört man den Vorteil von Parametern. Parameter sorgen dafür, dass ein Block während des Programmablaufs mit ganz verschiedenen Eingaben aufgerufen werden kann und richtig funktioniert.

8.3.8. Verwechslung von Index und Wert eines Listenelements

Wenn wir auf einzelne Elemente einer Liste zugreifen wollen, verwenden wir eine Zählvariable, z.B. innerhalb einer Zählschleife. Sei diese Zählvariable i. Dann liefert `i` die Ordnungszahl des aktuellen Elements, also den Platz, an dem es innerhalb der Liste steht. Den Wert des Elements, das an dieser Stelle in der Liste data steht, erhalten wir mit `item i of data`. Bei Sortierverfahren ist die Bedingung `if i > i+1` nie erfüllbar, da jede Zahl kleiner ist als die Summe der Zahl und Eins. Wenn man die Elemente vergleichen will, muss es heißen `if item i of data > item i+1 of data`.

8.3.9. 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 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.

Die Reporter, die am häufigsten als Anweisungen missbraucht werden, sind `join(...)` und [split ... by ....

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. `Stack.istEmpty`. Auch sie können nicht allein in einer Zeile stehen.

8.3.10. 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 als Referenz weiter. In der Folge wird also eine Liste durch verschiedene Variable angesprochen. Änderungen in der einen Liste tauchen automatisch in der anderen auf.

Abbildung 8.6.: Listen mit gleicher Referenz

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 Addbefehle auch vom Original unterscheiden kann. Alternativ kann man den `MAP`-Befehl ohne weitere Parameter verwenden: `set 2ndList to map ringify([ ]) over myList`.

8.3.11. Falsche Operationen bei Stapel und Schlange

Die Operationen der Klassen Stapel und Schlange sind eng definiert. So lässt sich

ein Stapel nur mit `Stack()`, `isEmpty()`, `top()`, `push(Inhalt)` und `pop()` ansprechen, eine Schlange nur mit `Queue()`, `isEmpty()`, `head()`, `enqueue(inhalt)` und `dequeue()`. Auf keinen Fall dürfen Listenbefehle mit Stapeln oder Schlangen verwendet werden! Die häufigsten Fehler sind `item i of Stapel/Schlange` und `length of Stapel/Schlange`.

8.3.12. Dynamische Reihung

Wenn in der Kopfzeile eines Blocks in der Parameterliste oder als Ergebnistyp bei einem Reporter **Dynamische Reihung** angegeben ist, dann müssen auch die entsprechenden Befehle aus der Anlage 1 Hilfsmittel für Prüflinge zur Verwendung in der schriftlichen Abiturprüfung im Fach Informatik verwendet werden. Es gibt also kein `item i of data`, sondern nur ein `data.getItem(i)` usw. Außerdem hat das erste Element den Index 0 und nicht 1. Es wird deshalb dringend davon abgeraten, diesen Datentyp zu verwenden. Statt dessen macht es Sinn, mit einem Satz wie **„Für die Implementierung der Dynamischen Reihung verwende ich eine Snap!-Liste“** darauf hinzuweisen, dass hier die „normalen“ Listenbefehle verwendet werden und das erste Element den Index 1 hat.

A. Anhang

A.1. 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:

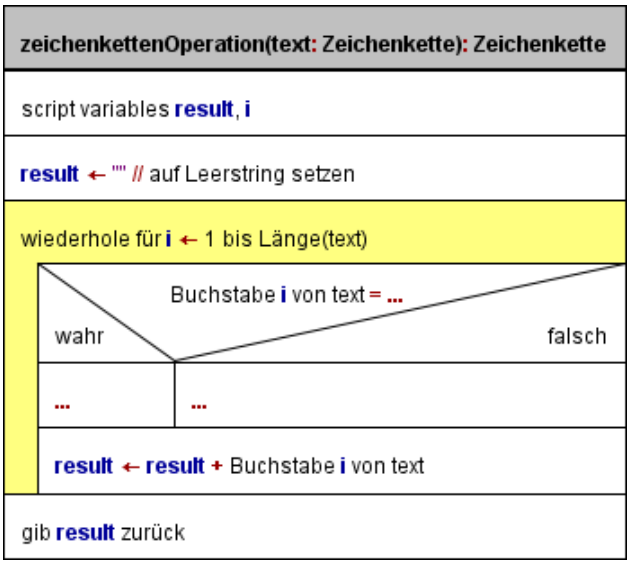


Abbildung A.1.: Grundrezept Operation mit Zeichenketten

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.