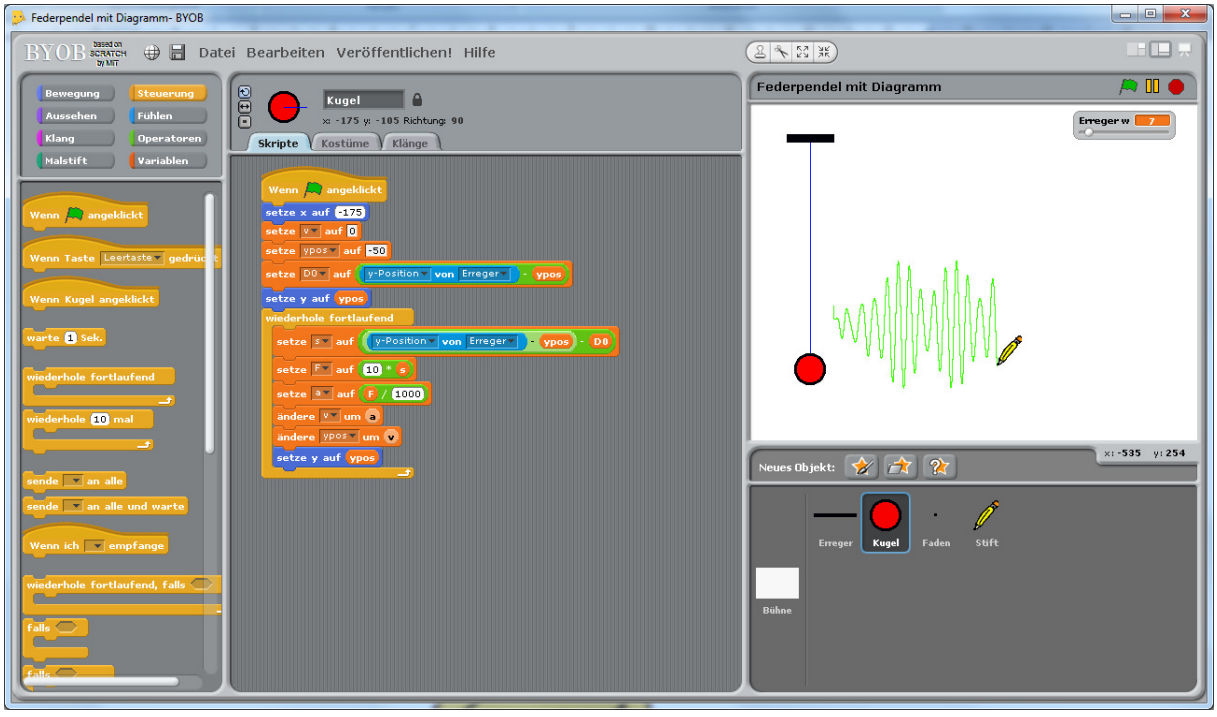
Eckart Modrow

Informatik mit BYOB / Snap!



© Eckart Modrow 2013
emodrow@informatik.uni-goettingen.de

Vorwort

Das vorliegende Skript stellt eine Sammlung von Programmierbeispielen dar, die insgesamt die Möglichkeiten der immer noch weitgehend unterschätzten grafischen Sprache BYOB aufzeigt. Ich halte die oft gehört Aussage „Über Scratch und BYOB zu <der richtigen Sprache>“, um welche es sich auch immer handeln mag, für nicht so richtig hilfreich, weil ich bisher keine für mich schlüssige Begründung dafür gehört habe, Algorithmen ein zweites Mal textuell zu implementieren, wenn diese selbst schon in einer grafischen Sprache entwickelt und erprobt worden sind. Das gilt vor allem, wenn in den „richtigen Sprachen“, gemeint sind meist Java oder Delphi, „Probleme“ gelöst werden, die in BYOB nur wenige kombinierte Blöcke erfordern. Statt einen großen Teil des Unterrichts mit Syntaxproblemen zu verbringen, schiene mir diese Zeit besser angelegt, wenn informatische Probleme bearbeitet würden, die insgesamt die Breite der Informatikanwendungen zeigen.

Natürlich lassen sich mit den „richtigen Sprachen“ Probleme ganz anderer Größenordnung behandeln als mit BYOB. Ich habe aber leise Zweifel, dass die Mehrzahl der Lernenden jemals zur Entwicklung „richtig großer“ Anwendungen kommt. Schlimmer noch: wenn der Unterricht vorrangig auf Lernende ausgerichtet ist, die dieses Ziel erreichen wollen, dann werden „normale“ Schülerinnen und Schüler ausgegrenzt, von der Wahl dieses wunderbaren Faches abgehalten. Ich befürchte, dass die relativ geringe Zahl von Informatiklernenden teilweise auf diese Ursache zurückzuführen ist – und umgekehrt hoffe ich natürlich, dass sich durch die Wahl der erst in jüngster Zeit entstandenen Werkzeuge (Scratch, BYOB und Varianten) die Basis des Informatikunterrichts stark verbreitern lässt.

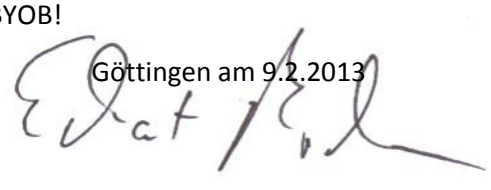
Die Auswahl der folgenden Beispiele ist relativ konservativ, lehnt sich teilweise noch eng an bestehenden Informatikunterricht an. Das ist Absicht. Ich hoffe, damit die unterrichtenden Kolleginnen und Kollegen vom traditionellen Unterricht „abzuholen“ und auf den Weg zu einem sehr an Kreativität, aber auch an der Vermittlung informatischer Konzepte ausgerichteten Unterricht mitzunehmen. Die ersten Beispiele sind sehr ausführlich und beschreiben detailliert den Umgang mit BYOB. In den späteren Kapiteln werden eher die Möglichkeiten der Sprache illustriert, am Ende ohne direkten Anwendungsbezug. Dieser Kompromiss ist dem Platzbedarf geschuldet, weil erweiterte Konzepte eigentlich auch erweiterte Problemstellungen erfordern.

Die Beispiele wurden fast durchgehend in BYOB 3.1.1 realisiert, weil Snap! zu diesem Zeitpunkt noch unvollständig ist. Soweit OOP-Verfahren fehlen, laufen die meisten Skripte aber auch in Snap! – und das wesentlich schneller.

Ich bedanke mich sehr bei Brian Harvey und Jens Mönig für ihre exzellente, konsequent an einem schlüssigen Konzept ausgerichtete Arbeit. Ich kenne kaum Beispiele für ein derart stringentes Vorgehen im Schulbereich. Die Lernenden werden es ihnen danken!

Ansonsten wünsche ich viel Freude bei der Arbeit mit BYOB!

Göttingen am 9.2.2013



Inhalt

1 Zu BYOB

1.1 Blockorientierte Sprachen

BYOB ist eine Abkürzung für *Build Your Own Blocks*. Damit ist schon ein Teil des Programms beschrieben: die Nutzer, bei denen es sich in erster Linie um Lernende an Schulen und Universitäten handeln wird, werden in die Lage versetzt, eigene *Blöcke* zu entwickeln. Man muss dazu wissen, dass praktisch alle Programmiersprachen *blockorientiert* sind: Befehlsfolgen lassen sich unter einem neuen Namen zusammenfassen. Den so entstehenden neuen *Befehlen* können bei Bedarf Werte (*Parameter*) übergeben werden, mit denen sie dann arbeiten, und sie können auch Ergebnisse zurückliefern. Damit gewinnen wir mehrere Vorteile:

Vorteile blockorientierter Sprachen

- **Programme werden kürzer**, weil Programmteile in die Blöcke ausgelagert und mehrfach verwendete Befehlsfolgen nur einmal geschrieben und dann unter dem neuen Namen mehrfach benutzt werden.
- **Programme enthalten weniger Fehler**, weil die Blöcke weitgehend unabhängig voneinander entwickelt und getestet werden und die aktuell entwickelte Befehlsfolge somit kurz und übersichtlich bleibt. „Lange“ Programmteile sind nur sehr selten notwendig und meist ein Zeichen für einen schlechten Programmierstil.
- **Programme erhalten einen eigenen Stil**, weil die neuen Befehle die Art spiegeln, in der die Programmierenden Probleme lösen.
- **Die Programmiersprache wird erweitert**, weil die erstellten Blöcke eine Befehle und somit auch neue Möglichkeiten repräsentieren.

1.2 Was ist BYOB?

die Entwickler

BYOB wurde (und wird) von Jens Mönig[1] und Brian Harvey[2] ab 2009 entwickelt und im Internet frei zur Verfügung gestellt[3]. Es basiert von der Oberfläche und dem Verhalten her auf *Scratch*[4], einer ebenfalls freien Programmierumgebung für Kinder, die am *MIT*[5] entwickelt wurde. Die umgesetzten Konzepte gehen allerdings weit darüber hinaus: hier liegen die Wurzeln bei *Scheme*, einem Dialekt der *LISP*-Sprache, der seit langem am MIT als Lehrsprache eingesetzt wird. Eingeführt werden sie z. B. in einem berühmten Lehrbuch von Harold Abelson sowie Gerald und Julie Sussman[6].

kaum Syntaxfehler

BYOB ist eine grafische Programmiersprache: Programme (*Skripte*) werden nicht als Text eingegeben, sondern aus *Kacheln* zusammengesetzt. Da sich diese Kacheln nur zusammenfügen lassen, wenn dieses einen Sinn ergibt, werden „falsch geschriebene“

[1] http://www.chirp.scratchr.org/blog/?page_id=2
[2] http://www.eecs.berkeley.edu/~bh/
[3] http://byob.berkeley.edu/
[4] http://scratch.mit.edu/
[5] Massachusetts Institute of Technology, Boston
[6] Abelson, Sussman: Struktur und Interpretation von Computerprogrammen, Springer 2001

Programme weitgehend verhindert. BYOB ist deshalb weitgehend *syntaxfrei*. Völlig frei von Syntax ist es trotzdem nicht, weil manche Blöcke unterschiedliche Kombinationen von Eingaben verarbeiten können: stellt man diese falsch zusammen, dann können durchaus Fehler auftreten. Allerdings passiert das eher bei fortgeschrittenen Konzepten. Wendet man diese an, dann sollte man auch wissen, was man tut.

BYOB ist außerordentlich „friedlich“: Fehler führen nicht zu Programmabstürzen, sondern werden z. B. durch das Auftauchen einer roten Markierung um die Kacheln angezeigt, die den Fehler verursachten – ohne dramatische Folgen. Die benutzten Kacheln, zu denen auch die entwickelten Blöcke gehören, „leben“ immer. Sie lassen sich durch Mausklicks ausführen, sodass ihre Wirkung direkt beobachtet werden kann. Damit wird es leicht, mit den Skripten zu experimentieren. Sie lassen sich testen, verändern, in Teile zerlegen und wieder gleich oder anders zusammensetzen. Wir erhalten damit einen zweiten Zugang zum Programmieren: neben der Problemanalyse und dem damit verbundenen *top-down*-Vorgehen tritt die experimentelle *bottom-up*-Konstruktion von Teilprogrammen, die zu einer Gesamtlösung zusammengesetzt werden können.

zwei Programmierstile

BYOB ist anschaulich: sowohl die Programmabläufe wie die Belegungen der Variablen lassen sich bei Bedarf am Bildschirm anzeigen und verfolgen.

anschaulich und erweiterbar

BYOB ist erweiterbar: durch die implementierten LISP-Konzepte lassen sich neue Kontrollstrukturen schaffen, die z. B. auf speziellen Datenstrukturen arbeiten.

BYOB ist objektorientiert, sogar auf unterschiedliche Weise: Objekte lassen sich sowohl über das Erschaffen von Prototypen mit anschließender Delegation wie auf unterschiedliche Art über Klassen erzeugen.

objektorientiert

BYOB ist erstklassig: alle benutzten Strukturen sind *first-class*, lassen sich also Variablen zuweisen oder als Parameter in Blöcken verwenden, können das Ergebnis eines Blockaufrufs sein oder Inhalt einer Datenstruktur. Weiterhin können sie unbenannt (*anonym*) sein, was für die implementierten Aspekte des Lambda-Kalküls, der Basis von LISP, wichtig ist. Folgerichtig trägt *Alonzo*, das Maskottchen von BYOB, ein stolzes Lambda als Tolle in seinem Haarschopf.

Alonzo

1.3 Was ist BYOB nicht?

die Grenzen

BYOB ist kein Produktionssystem. Es ist eine Technologiestudie im Auftrag des amerikanischen Bildungsministeriums im Rahmen von CE21 (Computing Education for the 21st Century), die u.a. zur Verringerung der Abbrecherquote in den technischen Fächern dienen soll. Es ist ein Werkzeug, um informatische Konzepte exemplarisch zu implementieren und zu erproben.

BYOB ist nicht schnell, ganz im Gegenteil - effizient ist es nicht.

BYOB ist nicht universell. Es dient in erster Linie zur Arbeit auf dem Gebiet der Algorithmen und Datenstrukturen. Wesentliche Bereiche der Informatik wie der Zugriff auf Dateien und das Internet, eine schnelle Bildbearbeitung usw. fehlen noch – auch

wenn ein einfacher Kommunikationsweg zwischen vernetzten Computern implementiert wurde.

BYOB ist nicht fehlerfrei. Als *Studie* ist es nicht völlig ausgetestet, sondern enthält einige Macken, z. B. beim Umgang mit lokalen Listen.

BYOB ist nicht fertig. Es wird laufend daran gearbeitet – mit allen damit verbundenen Konsequenzen für die Lehre.

1.4 Versionen von BYOB

Version 3.1.1

Anbindung technischer Geräte

Netzwerkzugriff

BYOB liegt seit Mitte 2011 in der stabilen Version 3.11 vor. Diese beruht technisch auf Scratch 1.4 und enthält damit allen Möglichkeiten, die dieses System bietet. Dazu gehört vor allem die Fähigkeit, auf externe technische Systeme zuzugreifen, also auf LEGO-WeDo-Geräte und das Picoboard. In Schulen sind die Motivationseffekte durch Beispiele aus dem Bereich *Steuern und Regeln*, die damit realisierbar sind, kaum zu überschätzen. Weiterhin bietet der einfache Zugriff auf andere Rechner (*Mesh*) zahlreiche Möglichkeiten, Probleme aus dem Bereich der *Vernetzung* entsprechend einfach zu behandeln. Inzwischen gibt es modifizierte BYOB-Versionen z. B. für die Steuerung von LEGO-Robotern[7], die einerseits die Möglichkeiten von BYOB erben, andererseits den technischen Bezug noch ausbauen.

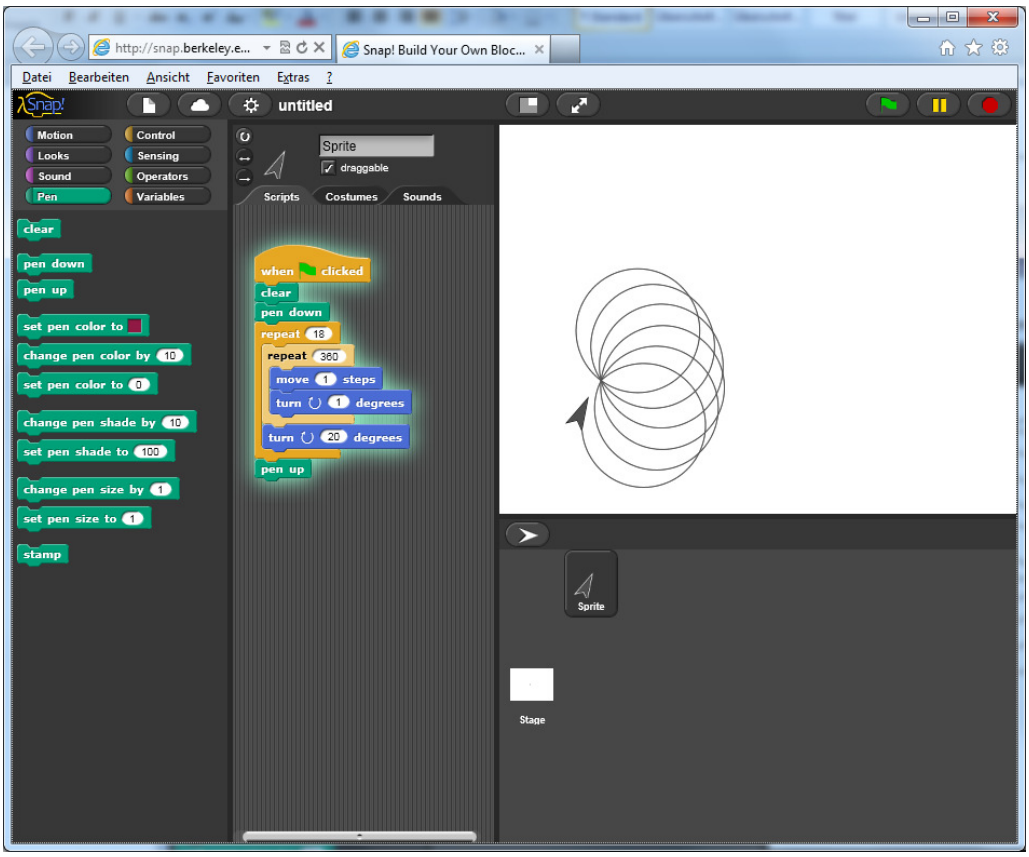
BYOB erbt nicht nur Eigenschaften von *scratch*, sondern sieht – absichtlich – auch sehr ähnlich aus (s. Titelseite). Diese kindgerechte Oberfläche hat nicht nur Vorteile: etwas ältere Lernende fühlen sich oft „kindlich“ behandelt, wenn sie mit diesem Werkzeug in die Programmentwicklung eingeführt werden. Die so erzeugten Widerstände können sich zu einem echten Lernhindernis entwickeln, da Neuanfänger noch nicht in der Lage sind, die sehr weitreichenden Konzepte von BYOB zu würdigen.

Da der Begriff *BYOB* auch noch sehr viel anders als für eine Programmiersprache verwendet wird, wurde BYOB in der neuen Version umgetauft zu *Snap!*. Snap! ist eine komplette Neuentwicklung, die statt in *Smalltalk* in *Javascript* geschrieben wird. Folgerichtig läuft es im Browser, erfordert also keine Installation. Snap! sieht sehr viel „professioneller“ aus als BYOB, zeigt anfangs nicht mehr Alonzo mit seiner Haartolle, sondern einen „coolen“ Pfeil als einziges Sprite. Die Oberfläche wirkt aufgeräumter.

Es kommen einige Erweiterungen neu hinzu, auf die wir später noch eingehen werden. Es gibt aber auch Verluste: mit der technischen Abkoppelung von *scratch* geht (in der derzeitigen Version) auch die Anbindung der LEGO-Komponenten verloren. Dafür soll die neue Version etwa 10-mal schneller als die alte sein.

Ich gehe davon aus, dass jedenfalls für einige Zeit die ältere BYOB-Version parallel zur neuen eingesetzt wird, wenn der Vorteil der Anbindung technischer Geräte die Nachteile der fehlenden Neuheiten überwiegt.

[7] https://launchpad.net/enchanting/



Die Oberfläche von Snap! bei laufendem Skript.

1.5 Der BYOB-Bildschirm

Der BYOB –Bildschirm besteht unterhalb der Menüleiste aus sechs Bereichen:

die Bildschirmbereiche

- Ganz links befinden sich die Befehlsregister, die in die Rubriken *Motion*, *Looks*, *Sound* usw. gegliedert sind. Klickt man auf den entsprechenden Knopf, dann werden unterhalb die Kacheln dieser Rubrik angezeigt. Passen sie nicht alle auf den Bildschirm, dann kann man in der üblichen Art den Bildschirmbereich scrollen.
- Rechts davon, also in der Mitte des Bildschirms, werden oben der Name des aktuell bearbeiteten Objekts – in BYOB *Sprite* genannt – sowie einige seiner Eigenschaften angezeigt. Den voreingestellten Namen des Sprites kann – und sollte – man hier ändern.

Sprite-bezogene Einstellungen

- Darunter befindet sich ein Bereich, in dem sich je nach Reiterkarte die *Skripte*, *Kostüme* und *Klänge* des Sprites bearbeiten lassen.
- Rechts-oben befindet sich das Ausgabefenster, in dem sich die Sprites bewegen. Dieses kann mithilfe der darüber befindlichen Buttons in seiner Größe verändert werden.
- Rechts-unten werden die zur Verfügung stehenden Sprites angezeigt. Klickt man auf eines, dann wechselt der mittlere Bereich zu dessen Skripten, Kostümen oder Klängen – je nach Auswahl.

die Menüleiste

- Die Menüleiste selbst bietet links die üblichen Menüs zum Laden und Speichern des Projekts sowie einzelner Sprites. Weiterhin können eine Reihe von Einstellungen vorgenommen werden, die sich in BYOB und Snap leicht unterscheiden (s. Register). Eine Möglichkeit besteht darin, die Sprache einzustellen. Ich empfehle derzeit, bei der englischen Version zu bleiben, da bei anderen Versionen ggf. Inkonsistenzen bei der Übersetzung auftreten können.

- Ganz rechts finden wir die aus Scratch bekannte grüne Flagge, mit der bei Verwendung des entsprechenden Blocks mehrere Skripte gleichzeitig gestartet werden können. Der Pause-Knopf daneben lässt entsprechend alles pausieren und der rote Knopf beendet alle laufenden Skripte. Einzelne Skripte oder Kacheln startet man einfach durch anklicken.

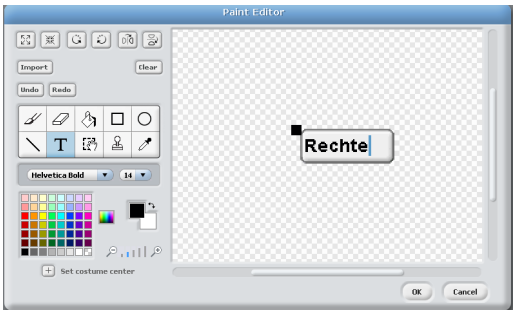
2 Ein Zeichenprogramm

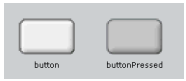
Wir beginnen mit einem kleinen Beispiel, das die Ereignisverarbeitung und den direkten Umgang mit einfachen Skripten demonstrieren soll. Mithilfe von Knöpfen sollen verschiedene Funktionen eingestellt werden können, die von verschiedenen Gruppen realisiert werden. Das Beispiel stellt Möglichkeiten zu arbeitsteiligem Vorgehen sowie zur Fehlersuche vor, an die sich die Lernenden früh gewöhnen sollten.

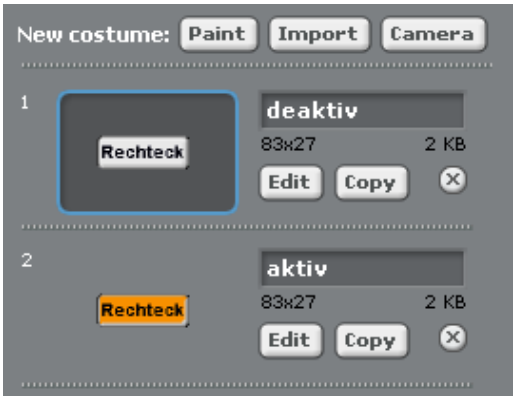
2.1 Mit Knöpfen arbeiten

Sprites erzeugen

Als erstes löschen wir Alonzo, so schön er auch sein mag. Stattdessen benötigen wir mehrere Knöpfe um Einstellungen vorzunehmen. In den Kostümvorlagen für *Sprites* (BYOB-Objekte) finden wir unter *Things* zwei geeignete Kostüme. Das erste nehmen wir und erzeugen damit ein neues Sprite, das wir B-Rechteck taufen. Wir gehen im Arbeitsbereich auf dessen Kostüme und bearbeiten das vorhandene (*edit*). Im nun geöffneten Zeichnungseditor wählen wir das Textwerkzeug und beschriften den Knopf.







Danach kopieren wir das Kostüm und bearbeiten dieses, indem wir etwas Farbe in den Knopf gießen. Dadurch soll der aktive Zustand visualisiert werden. Wir wählen geeignete Kostümnamen.

Zuletzt kopieren wir diesen Button (rechter Mausklick im Spritebereich) und erzeugen so drei weitere Buttons für Kreise und die Farben Rot und Schwarz. Diese benennen wir mit B-Kreis, B-Rot und B-Schwarz, beschriften sie geeignet und ordnen sie auf dem Bildschirm an.

auf Mausklicks reagieren

Klicken wir einen der Knöpfe an, dann soll er seinen durch das Kostüm angezeigten Zustand wechseln – und der zugehörige zweite soll ggf. deaktiviert werden. Wie macht man das?

Wir versuchen es zuerst mit einem Kostümwechsel.

wenn B-Rechteck angeklickt wurde
wechsele das Kostüm

Wir finden die Kachel next costume aus der Kategorie looks, und in der Kategorie control eine *Hut*-Kachel (hat) mit der richtigen Beschriftung. „Hüte“ dienen dazu, Skripte unter bestimmten Bedingungen zu starten – und genau das wollen wir ja.

Klicken wir jetzt den Knopf B-Rechteck an, dann wechselt er wie gewünscht sein Aussehen. Der zugehörige Knopf B-Kreis aber noch nicht. Wir ergänzen deshalb unser *Struktogramm*, damit B-Kreis deaktiviert wird, falls B-Rechteck aktiviert wurde.

wenn B-Rechteck angeklickt wurde	
wechsele das Kostüm	
die Kostümnummer ist 2 wahr	falsch
sorge dafür, dass B-Kreis deaktiviert wird	

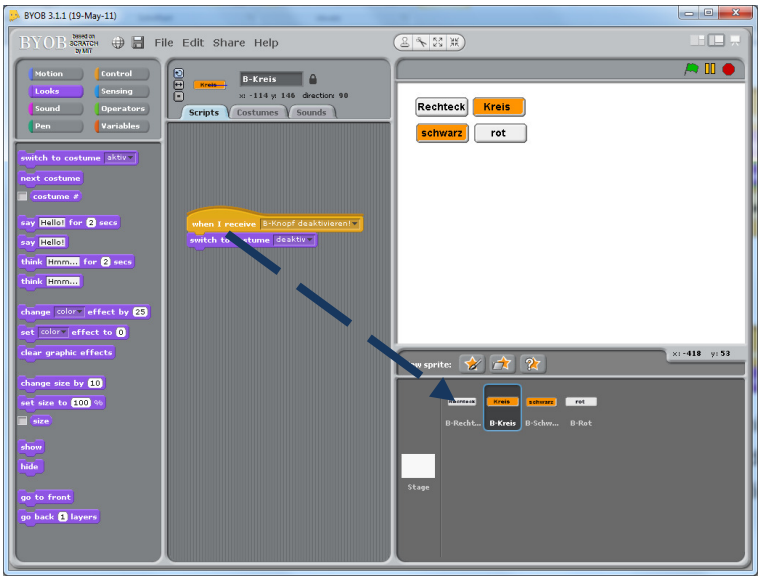
Botschaften einsetzen

Jetzt haben wir ein Problem: Wir möchten, dass ein *anderes* Sprite arbeitet, nicht das gerade aktive! So etwas kann man durch *Botschaften* regeln. Das aktive Sprite sendet eine Botschaft aus und das adressierte reagiert auf diese, wenn eine entsprechende Hat-Kachel eingesetzt wird. Diese Kacheln finden wir im Control-Bereich.[8]

Der B-Knopf muss natürlich auf diese Nachricht (*message*) reagieren.

Skripte übertragen

Jetzt müssen die anderen Knöpfe mit einer entsprechenden Funktionalität ausgestattet werden. Wir ziehen dazu die gewünschten Skripte (hier: beide) aus dem Skriptbereich eines Sprites auf diejenigen Sprites im Spritebereich, auf die sie übertragen werden sollen. Danach ändern wir die entsprechenden Werte (Botschaften, …).



Nach diesen Ergänzungen arbeitet die Knopfgruppe wie gewünscht: es kann höchstens einer der Knöpfe B-Rechteck und B-Kreis bzw. B-Rot und B-Schwarz ausgewählt werden und das Ergebnis wird visualisiert.

[8] Das genaue Vorgehen zum Gestalten von Skripten wird unter 2.2 beschrieben.

Kommentare einfügen

Ein gutes Zeichenprogramm ist ziemlich umfangreich, wenn wir uns nicht nur auf zwei Farben und zwei Formen beschränken. Wir sollten deshalb arbeitsteilig in verschiedenen Gruppen arbeiten und die Ergebnisse später zusammenstellen. Da die Skripte später von anderen gelesen und verstanden werden müssen, ist es gute Praxis, *Kommentare* an die Skripte anzuhängen. Dazu machen wir einen Rechtsklick in den Skriptbereich und erzeugen mit add comment einen neuen Kommentar. Diesen können wir mit der Maus verschieben. Berührt er einen Befehlsblock, dann wird dieser weiß umrandet, um anzuzeigen, dass der Kommentar andocken kann. Beim Loslassen der Maustaste geschieht dieses. Bei Bedarf können wir Kommentare durch Anklicken des kleinen Pfeils verkleinern.

2.2 Einfache Algorithmik und Fehlersuche

Die Knöpfe müssen mit Funktionalität versehen werden. Wir beschränken uns dabei zuerst auf die Rechtecke, bauen dabei die Ereignisverarbeitung aus und demonstrieren den Umgang mit einfachen Skripten. Weiterhin werden verschiedene Möglichkeiten zur Fehlersuche vorgestellt, an die sich die Lernenden früh gewöhnen sollten.

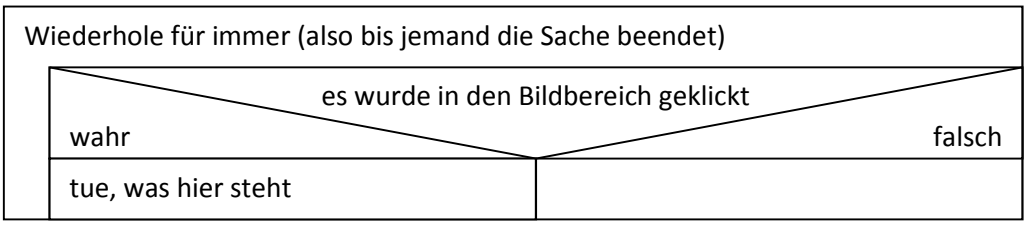
2.2.1 Koordinaten anzeigen

Unser Ziel soll es hier sein, mit der Maus zweimal auf den Bildschirm zu klicken, sodass danach ein Rechteck zwischen den angeklickten Punkten gezeichnet werden kann. Dazu sollen zuerst einmal die Koordinaten des angeklickten Punkts angezeigt werden.

Wir klicken in den Bildbereich[9] - nichts passiert.

Der Grund ist einfach: niemand beachtet diese Mausklicks. Wir benötigen deshalb eine *Kontrollanweisung* aus der Rubrik Control, die dafür sorgt, dass fortlaufend nachgesehen wird, ob jemand in den Bildbereich mit der Maus klickt. Beschrieben werden solche Abläufe wie schon oben durch *Struktogramme*, die sich aus den elementaren Kontrollstrukturen *Wiederholung*, *Alternative* und *Sequenz* zusammensetzen lassen. Hier brauchen wir erstmal nur zwei davon.

eine Wiederholungsanweisung



[9] also den Bereich rechts-oben

Kontrollanweisungen finden wir in der Control-Rubrik, von der rechts ein Ausschnitt gezeigt wird. Dort müssen wir suchen. Wir finden

Control-Rubrik

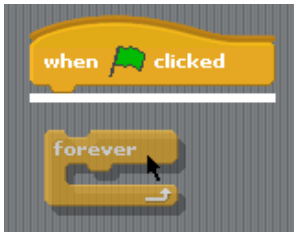
- einen (von mehreren) „Hut“ – die hat-Kachel mit der grünen Flagge. Alle Befehlskacheln, die an diese andocken, werden gestartet, wenn rechts-oben über dem Bildbereich die grüne Startflagge angeklickt wird,

- eine (von mehreren) *Wiederholungsanweisungen*[10], deren Beschriftung hoffen lässt, dass sie das Gewünschte bewirkt

- und eine *einfache Alternative*, die anhand eines *Prädikats*, das wahr oder falsch sein kann, auswählt, ob die noch einzufügenden Blöcke ausgeführt werden.

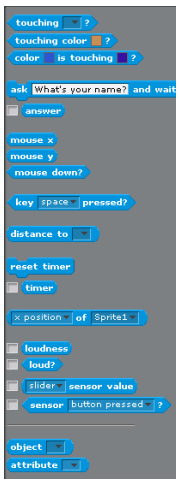
Ein Prädikat (oder eine Bedingung) erkennt man in BYOB an seiner Rautenform. Leerstellen, in die Prädikate eingefügt werden können, haben ebenfalls diese Form. Andere Blöcke kann man an dieser Stelle nicht einfügen.[11]

Jetzt müssen diese Blöcke zusammengefügt werden – so, wie es das *Struktogramm* angibt. Schiebt man eine Kachel mit der Maus in die Nähe einer Andockstelle, dann leuchtet diese weiß auf. Lässt man die Maustaste dann los, kann rastet die Kachel an der markierten Stelle ein. Gleichfarbige Kacheln werden dabei farblich leicht unterschieden[12], um die Lesbarkeit zu erhöhen.

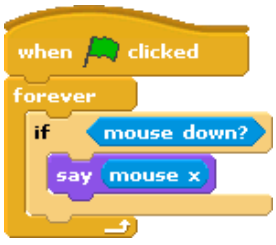


Auf diese Art können wir dann die drei Kontrollkacheln zu einem Skript zusammenfügen.

Sensing-Rubrik



Wir benötigen jetzt noch das Prädikat, also die Bedingung, dass mit der Maus in den Bildbereich geklickt wurde. Diese finden wir in der Kategorie Sensing als mouse down? Direkt darüber finden wir zwei *Reporter*-Blöcke, die die Mauskoordinaten ermitteln: mouse x und mouse y. Wir lassen die x-Koordinate durch Alonzo anzeigen. Dazu wählen wir den Befehl say aus der Rubrik Looks. In diesen lassen wir den richtigen Reporterblock einrasten.

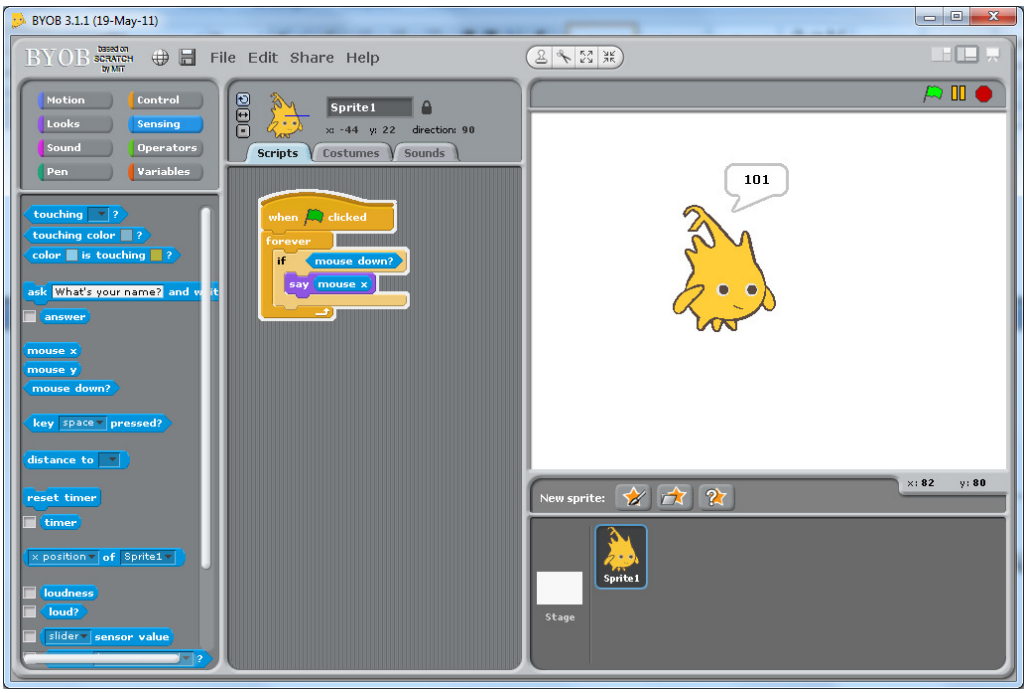


[10] oft auch „Schleifen“ genannt

[11] So ganz stimmt das nicht: wenn ein Block oder eine andere Größe ebenfalls einen Wahrheitswert ergeben kann, dann klappt es auch mit dem Einfügen.

[12] wenn die Option „Zebra Coloring“ eingeschaltet ist

Klicken wir jetzt auf die grüne Startflagge, dann erzählt uns Alonzo, was wir wissen wollen. (Man erkennt am weißen Rand, dass das Skript gerade läuft.)



2.2.2 Aufgaben

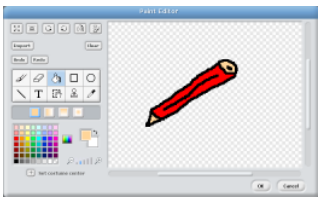
Um die folgenden Aufgabe zu bearbeiten, müssen Sie sich etwas in den Befehlsregistern umsehen! Experimentieren Sie mit den Blöcken!

1. a: Lassen Sie Alonzo beide Mauskoordinaten nacheinander anzeigen.
   b: Lassen Sie ihn bei Mausklicks den Abstand der Maus von sich selbst anzeigen.
   c: Lassen Sie ihn dieses kontinuierlich anzeigen.
   d: Lassen Sie ihn dieses anzeigen, wenn die Maustaste nicht gedrückt ist.
   e: Lassen Sie Alonzo sagen, ob die Maustaste gedrückt wurde oder nicht.

2. a: Zeichnen Sie für Alonzo ein neues Kostüm, indem Sie das alte kopieren und umfärben. Sie können ihn auch mit einem kleinen Spitzbart versehen.
   b: Lassen Sie Alonzo das Kostüm wechseln, wenn die Maustaste gedrückt wird.
   c: Lassen Sie Alonzo das Kostüm wechseln, wenn die Leertaste auf der Tastatur gedrückt wird.
   d: Lassen Sie ihn das Kostüm wechseln, wenn die Maus „zu nahe“ an ihn heran kommt. Sobald sie sich entfernt, nimmt er wieder das alte Kostüm.

3. a: Wählen Sie für Alonzo neue Kostüme aus dem Dateibereich aus. Sorgen Sie dafür, dass er diese unter „geeigneten“ Bedingungen wechselt. Löschen Sie danach Alonzo (Rechtsklick auf das Symbol im Sprite-Bereich und auswählen).
   b: Laden Sie ein neues Sprite aus dem Dateibereich.
   c: Zeichnen Sie ein neues Sprite mithilfe des Grafik-Editors.

2.2.3 Koordinaten speichern

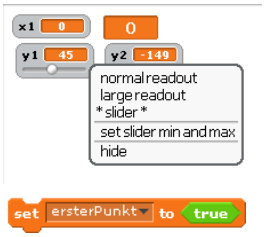


Wir ersetzen jetzt Alonzo durch einen schönen Malstift, den wir neu zeichnen.

Wollen wir Rechtecke zeichnen, dann benötigen wir die Koordinaten von zwei Punkten, also vier Zahlenwerte. Dazu müssen wir diese Werte zum richtigen Zeitpunkt speichern – wenn die Maustaste gedrückt wird. Geeignet dazu sind *Variable*, die wir in der Variables-Rubrik erzeugen und bearbeiten können. Wir wollen diese Variable in der üblichen Art mit x1, y1, x2 und y2 bezeichnen. Dazu klicken wir den Knopf mit der Beschriftung *Make a variable* an und geben im erscheinenden Eingabefeld den richtigen Namen ein. Dort können wir auch auswählen, ob es sich um eine *globale Variable*, die allen Sprites bekannt ist, oder eine *lokale Variable* handelt, die nur das betroffene Sprite kennt.

Variables-Rubrik

Wir erzeugen jetzt die vier Variablen auf diesem Wege und klicken bei Bedarf jeweils links neben ihnen die Checkbox an, um die Variablenwerte im Bildbereich anzuzeigen. Dabei gibt es verschiedene Anzeigeoptionen, die wir durch einen Rechtsklick auf die Variablendarstellung im Bildbereich auswählen können. Im Bild rechts sind diese dargestellt: wir können Variable *normal* darstellen (x1), *vergrößert* (x2), mit einem *Schieberegler* (y1), dessen *Grenzwerte* wir einstellen können (y2).



Jetzt müssen wir uns nur noch merken, welcher Eckpunkt beim Mausklick gespeichert werden soll – der erste oder der zweite. Wir führen dafür eine neue Variable ersterPunkt ein, der wir abwechselnd die Wahrheitswerte true oder false zuweisen, die wir in der Operators-Rubrik finden. Anfangs erhält sie den Wert true, indem wir den entsprechenden Zuweisungsblock set <variable> to <value> benutzen[13].

Operators-Rubrik

Damit können wir das Vorgehen verfeinern:

Wiederhole für immer

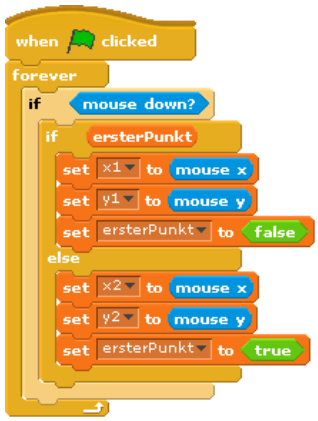
es wurde in den Bildbereich geklickt: wahr	es wurde in den Bildbereich geklickt: falsch
ersterPunkt?: wahr	ersterPunkt?: falsch
x1 ← x-Koordinate der Maus	x2 ← x-Koordinate der Maus
y1 ← y-Koordinate der Maus	y2 ← y-Koordinate der Maus
ersterPunkt ← falsch	Zeichne ein Rechteck mit den Eckpunkten (x1|y1) und (x2|y2)
	ersterPunkt ← wahr

[13] Die spitzen Klammern kennzeichnen die Parameter, also Werte, die beim Aufruf des Blocks angegeben werden müssen.

Wir testen das Programm zuerst nur mit dem Setzen der Variablenwerte. Das Zeichnen kommt später.

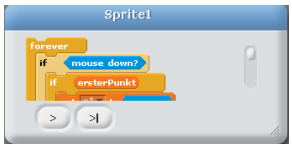
Wir ärgern uns, denn trotz aller Mühen erhalten jeweils die x- und y-Koordinaten der beiden Punkte immer den gleichen Wert.

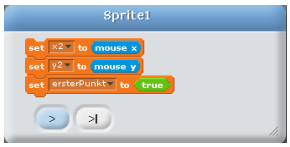
Was ist passiert?



Debuggen …

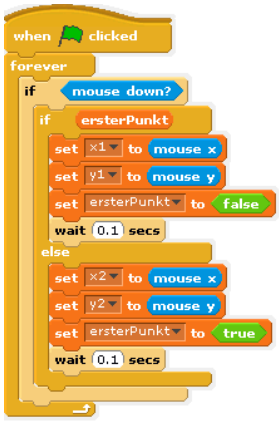
Wir versuchen erst zuerst mit dem *Fehlersuchmodus* und fügen dazu die entsprechende Kachel debug an verschiedenen Stellen in den Programmtext ein – z. B. ganz am Anfang vor „forever“. Es erscheint ein Meldungsfenster, das den aktuellen Skriptausschnitt anzeigt. Der linke Knopf (>) beendet das *Debuggen* (Fehlersuchen), der rechte (>|) zeigt den nächsten Schritt. In unserem Fall ist dieser Modus nicht sehr geeignet, weil wir zur Bedienung die Maus benutzen – und die brauchen wir eigentlich im Bildschirmbereich. Wir verschieben deshalb die Debug-Kachel in die Alternative, sodass sie nur dann aufgerufen wird, wenn vorher die Maus geklickt wurde. Das klappt jetzt besser – es werden nur die Koordinaten von einem Punkt verändert!





Was ist der Unterschied zu vorher?

Die Befehle werden jetzt mit großem zeitlichem Abstand ausgeführt, eben nur dann, wenn der entsprechende Knopf betätigt wird. Es sieht so aus, als ob der Computer ohne diese Pausen wesentlich schneller ist als unsere Hand: wenn nach einem Mausklick der nächste Schleifendurchlauf beginnt, ist die Maustaste immer noch gedrückt. Wir müssen die Abläufe verzögern, indem wir Pausen einprogrammieren – mit dem entsprechenden wait-Block aus der Control-Kategorie. Eine Zehntelsekunde genügt dafür.



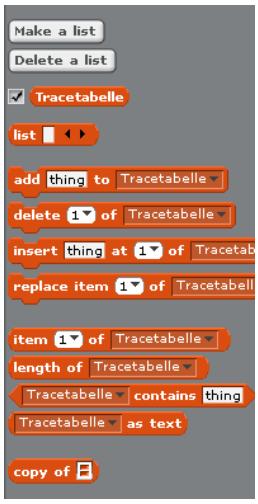
… und andere Testmöglichkeiten

Wir können daraus lernen, dass Programme selbst dann nicht funktionieren müssen, wenn sie logisch richtig formuliert wurden. Der Computer führt aus, was in den Skripten steht, nicht, was damit gemeint war. Wir müssen unsere Skripte in jedem Fall testen, möglichst in kleinen Portionen, damit neue Fehlerquellen sich nur in den jeweils neu hinzugefügten Befehlen verstecken können, und bei denen sollte es sich nur um jeweils wenige handeln. Als Testmöglichkeiten stehen uns neben dem Debugger-Modus Kontrollausgaben (von Variablen und/oder Meldungen des Sprites) auch Zeitverzögerungen zur Verfügung, die durch Pausen bei der Ausgabe (say <message> for <number> secs) oder das Pausenkommando (wait <number> secs) verursacht werden können.

2.2.4 Tracetabellen erzeugen

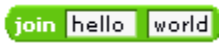
Eine gute Möglichkeit Programme zu überprüfen, ist der Einsatz von *Tracetabellen*. Dafür wird tabellarisch notiert, welche Variablen ihre Werte ändern. In der Tabelle verläuft die Zeitachse dann von oben nach unten. In BYOB können wir dafür leicht eine *Liste* einsetzen, die wir in der Variables-Rubrik ähnlich wie andere Variablen erzeugen können. Listen können viele gleiche oder unterschiedliche Dinge speichern. Sie sind so etwas wie universale Container. In unserem Fall soll die Liste Tracetabelle heißen.

Listen bearbeiten



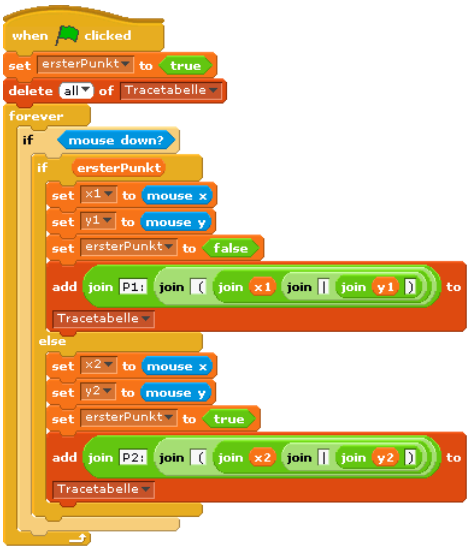
Alle Elemente in einer Liste können wir mit delete <all> of Tracetabelle löschen, neue Elemente werden mit add <thing> to Tracetabelle hinzugefügt.

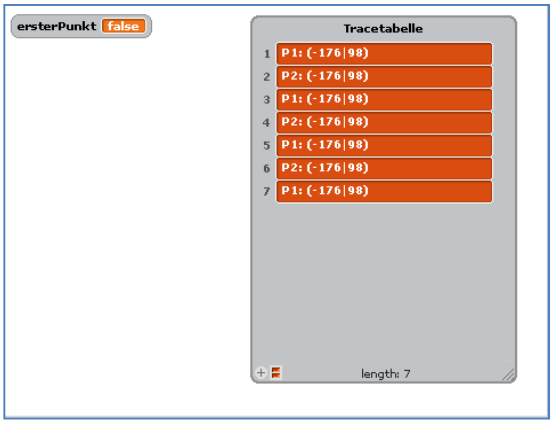
Was wird hinzugefügt?

In der Operators-Rubrik finden wir die Kachel 

, mit der sich zwei Elemente zu einer Zeichenkette zusammensetzen lassen. Solche Zeichenketten werden dann der Tracetabellen-Liste hinzugefügt. Wollen wir mehr als zwei Dinge zu einer Zeichenkette zusammenfügen, dann müssen wir mehrere join-Kacheln schachteln[14]. In unserem Fall wollen wir Punktkoordinaten schön mit Klammern und durch senkrechte Striche getrennt erzeugen: 

Solche Befehle fügen wir jetzt in unser Skript ein und betrachten das Ergebnis nach einem einzigen Mausklick.



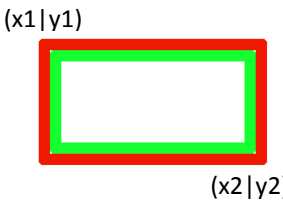


Man kann gut nachvollziehen, wie schnell die Wertzuweisungen an die Punktkoordinaten gewechselt haben – und auch darin den Fehler im ersten Skript erkennen.

[14] In Snap! kann man durch Erweiterung der Eingabefelder die Schachtelung vermeiden.

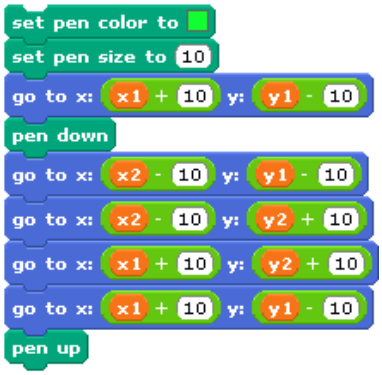
2.2.5 Rechtecke zeichnen

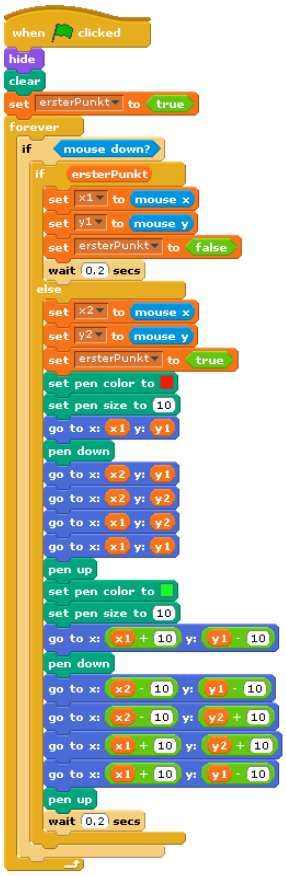
ein schönes Rechteck



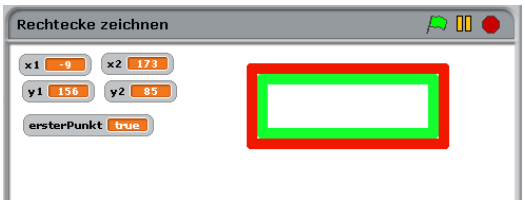
Mittlerweile ist es uns gelungen, die Koordinaten von zwei Bildschirmpunkten (x1|y1) und (x2|y2) durch Mausklicks zu bestimmen. Zwischen denen zeichnen wir jetzt ein besonders schönes Rechteck, das einen breiten roten Rand haben soll und darin einen weiteren grünen. Der Innenraum bleibt weiß.

Der äußere Rahmen ist schnell erstellt: Strichbreite und Farbe einstellen und dann die Eckpunkte mit abgesenktem Stift anfahren. Der innere Rand ist auch leicht zu zeichnen: wir gehen fast genauso vor, fahren dabei aber um 10 Pixel nach innen versetzte Punkte an.





Das Ergebnis ist wie erwartet - erstmal.



Wir sollten uns aber nicht zu früh freuen, sondern das Programm systematisch testen. Dazu klicken wir Punkte an, die

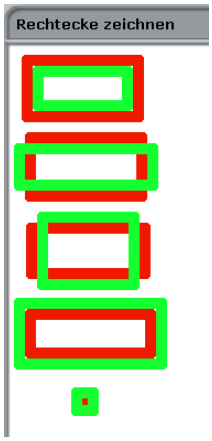
a) oben-links und unten-rechts

b) oben-rechts und unten-links

c) unten-links und oben-rechts

d) unten-rechts und oben-links

e) sehr nahe beieinander



im Rechteck liegen. Die Ergebnisse sind abgebildet. Sie entsprechen nicht so richtig den Vorgaben.

Obwohl unser Programm richtig arbeitet, wenn es wie vorgesehen benutzt wird, könnten andere Benutzer/innen es auch anders benutzen – und dann passieren die gut sichtbaren Fehler, die hier natürlich mit der Berechnung der Eckpunkte des „inneren“ grünen Randes zu tun haben. Diese muss also dringend verbessert werden.

Aus den gemachten Erfahrungen leiten wir einige Regeln für die Erstellung von Programmskripten ab:

Skripte sollten

- durch ausreichend viele Testläufe erprobt werden. Dabei sollten die Extremfälle (sehr große/kleine Werte, positive/negative/fehlerhafte Eingaben, …) systematisch durchgespielt werden.
- so weit möglich mit Testausgaben z. B. der Variablenwerte versehen werden, sodass ihr Ablauf leicht zu visualisieren und so zu überprüfen ist. Tracetabellen sind dafür ein gutes Mittel.
- in kleinen „Happen“ erstellt werden, die einzeln getestet werden können. Aus diesen wird dann das Gesamtskript zusammengesetzt.
- kurz sein. Sind sie das nicht, dann sollten sie in Teile aufgespalten werden. Das entwickelte Beispiel ist also schon viel zu lang.

2.3 Kreise zeichnen

Mit unseren Mausklicks wollen wir jetzt das Zeichnen von Kreisen in der ausgewählten Farbe starten. Die Kreise sollen erst einmal alle die gleiche Größe haben.

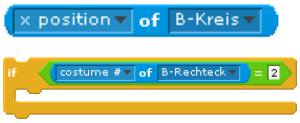
Einen Kreis können wir zeichnen, indem wir ein Sprite wiederholt ein Stückchen vor gehen lassen und danach um einen kleinen Winkel drehen. Hat man es eilig, dann kann man die Größen auch um 2, 3 oder mehr Schritte ändern und die *Zählschleife* entsprechend weniger oft durchlaufen lassen.

Wiederhole 360 mal
gehe einen Schritt vor
drehe um ein Grad

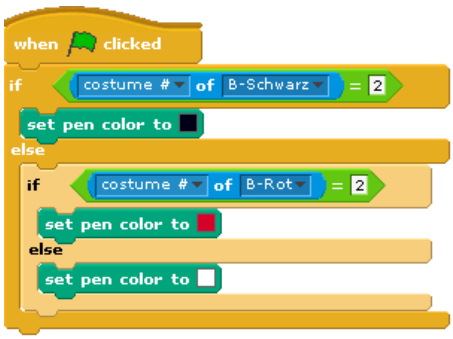
Wir müssen aber noch zwei andere Probleme lösen:

1. Wir wollen unseren Stift **entweder** Rechtecke (mithilfe von zwei Mausklicks) **oder** Kreise (mithilfe eines Klicks) zeichnen lassen.
2. Der Stift soll in der ausgewählten Farbe zeichnen.

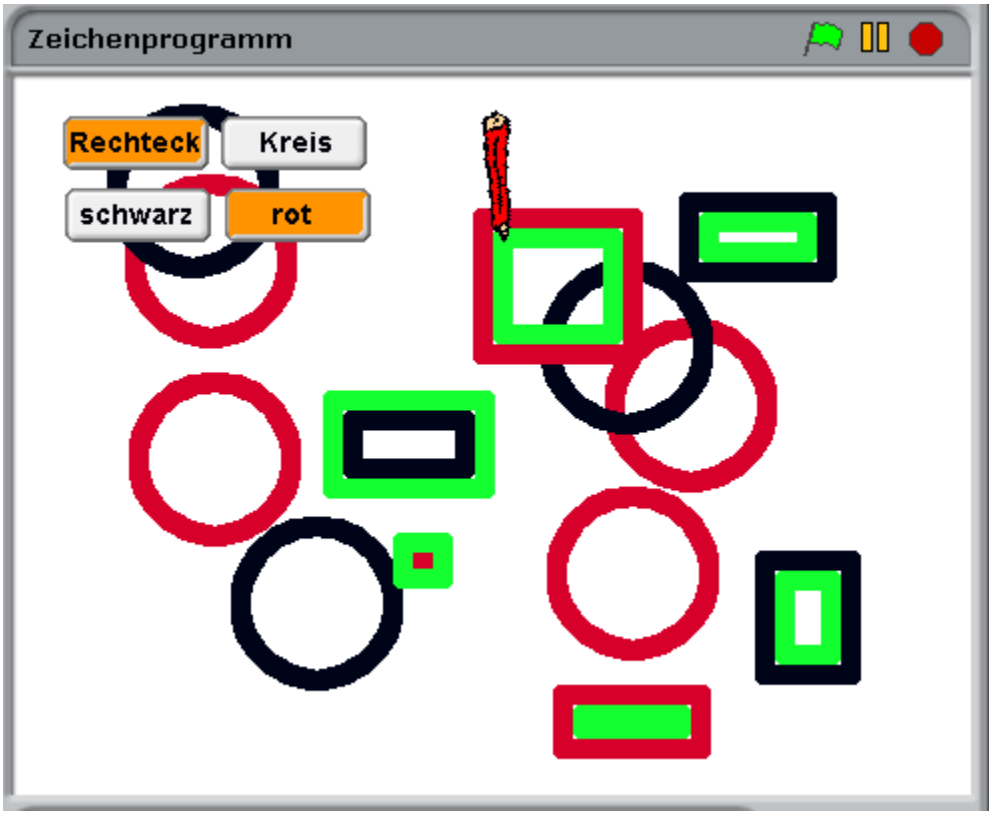
Zugriff auf Attribute anderer Sprites

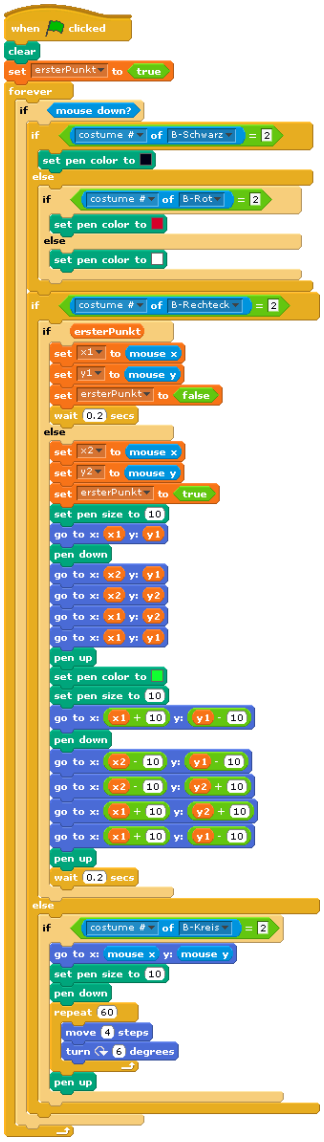


Wir müssen dafür die Knöpfe befragen, in welchem Zustand – ausgedrückt durch die Kostümnummer – sie sich befinden. Das gelingt mit einer Kachel aus der Sensing-Kategorie: <attribute> of <object>. Mit dessen Hilfe können wir ein Skript nur dann ausführen lassen, wenn die Kostümnummer eines Sprites den richtigen Wert hat. Wir demonstrieren das an der Wahl der Zeichenfarbe, wobei bei keinem aktivierten Knopf in Weiß gezeichnet wird.



Dieses Vorgehen übertragen wir jetzt auf das Zeichnen der Figuren- und erhalten ein ellenlanges Skript, was auf einen ziemlich katastrophalen Programmierstil hindeutet. Aber immerhin – es funktioniert.





Unser Programm besteht aus vier ziemlich klar getrennten Blöcken:

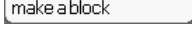
- der Bestimmung der Zeichenfarbe,
- der Bestimmung der Koordinaten,
- dem Zeichnen eines Rechtecks
- und dem Zeichnen eines Kreises.

Weshalb teilen wir es dann nicht entsprechend auf?

Genau das kann BYOB leisten, indem die entsprechenden Befehle in einen *Block* verschoben werden. Wir benutzen diesen als eine Art *Makro*, als Abkürzung für eine Befehlsfolge.

einen neuen Block erzeugen

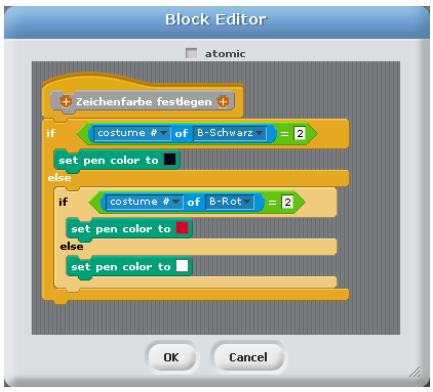
In der Variables-Kategorie finden wir einen Knopf Make a block, den wir dafür benutzen. Stattdessen können wir auch einen Rechtsklick auf den Skriptbereich ausführen und den entsprechenden Menüpunkt wählen. In beiden Fällen erschein ein Auswahlfenster, in dem wir den Blocknamen und seinen Gültigkeitsbereich (global oder lokal) festlegen, die Kategorie auswählen, in der er erscheinen soll (und die seine Farbe festlegt) und seinen Typ bestimmen: ein ausführbares Skript (*command*), einen *Reporter*-Block, der ein Ergebnis zurückgibt oder ein *Prädikat*, das einen Wahrheitswert ermittelt. Wir bestimmen erst einmal nur den Namen.

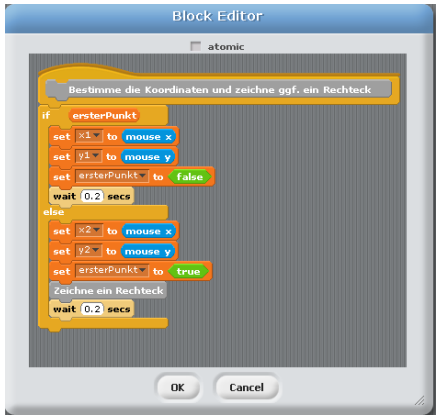


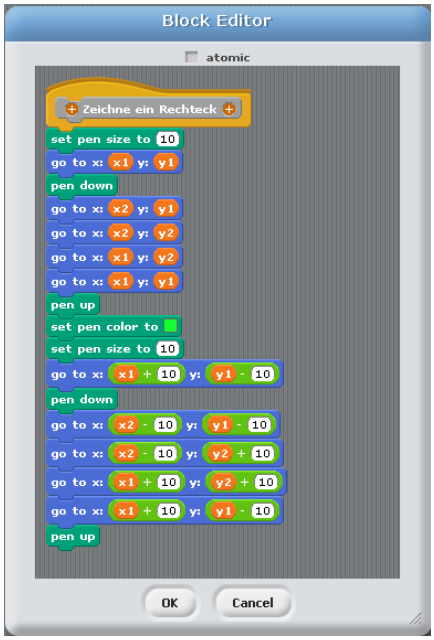
In den jetzt erscheinenden Blockeditor ziehen wir einfach die schon erprobte Befehlsfolge hinein. Fertig.

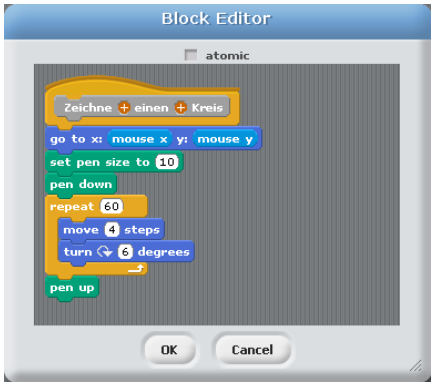
Entsprechend gehen wir mit den anderen drei Befehlsblöcken vor: wir erhalten neue Blöcke mit den frei gewählten Aufschriften Zeichenfarbe festlegen, Zeichne ein Rechteck, Zeichne einen Kreis sowie Bestimme die Koordinaten. Mit diesen liest sich das Zeichenprogramm schon wesentlich leichter, und da kurze Programme auch meist weniger fehleranfällig sind, hat diese Verkürzung auch einen „informatischen“ Sinn.

Blöcke als Makros einsetzen









2.4 Arbeitsteilig vorgehen

die Funktionalität aufteilen …

Wenn wir unser Zeichenprogramm etwas genauer ansehen, dann haben wir zwei Gruppen von Knöpfen für die Auswahl der Figuren und Farben, die beide noch dringend erweitert werden müssen, sowie einen Zeichenstift, der neu geschriebene Blöcke benutzt, die unabhängig voneinander entwickelt werden können. Das Projekt bietet sich also für arbeitsteiliges Vorgehen an: einige Gruppen entwickeln zuverlässig funktionierende Knopfgruppen, andere sorgen dafür, dass die bei den Mausklicks ermittelten Koordinaten wirklich im Zeichenbereich und nicht sonstwo liegen oder entwickeln einen Radierer, neue Figuren oder Farbverläufe und andere grafische Elemente.

… und in Blöcke verlagern

Eine Möglichkeit zur Arbeitsteilung haben wir schon kennengelernt: die Blöcke. Eine zweite wären zusätzliche Sprites, mit denen neue Möglichkeiten verbunden sind. Kommunizieren können diese einerseits über globale Variable (andere haben wir noch gar nicht benutzt), andererseits durch den Aufruf lokaler Methoden (das kommt später). Das alles funktioniert aber nur, wenn wir Variable, Blöcke und Sprites von einem Programm zum anderen transportieren können, weil arbeitsteiliges Vorgehen natürlich an verschiedenen Rechnern erfolgt.

Sprites exportieren und importieren

Blöcke und Variable sind an Sprites gebunden, globale Variable aber nur, wenn in einem Skript des Sprites ein Zugriff auf sie erfolgt. Wir wissen schon, dass man Sprites aus den Bibliotheken von BYOB *importieren* kann. Es geht also nur um die Frage, wie man Sprites *exportieren* kann. Das geschieht ganz einfach durch den entsprechenden Menüpunkt, den man entweder im File-Menü oder bei Rechtsklick auf ein Sprite erreicht. Wir können es dann ganz normal als Datei speichern – mit der Endung *ysp*. Andere BYOB-Programme importieren diese Sprites auf dem üblichen Weg mithilfe des Buttons  im Sprite-Bereich. Sie können das Sprite sogar wieder löschen, ohne dass die damit importierten globalen Blöcke und Variablen verloren gehen. Auf diese Weise können Sprites z. B. als Bibliotheken dienen, die die gleiche Rolle übernehmen wie in anderen Sprachen.[15]

Man löst ein Problem arbeitsteilig, indem man

- die Funktionalität auf verschiedene Blöcke aufteilt und diese arbeitsteilig entwickelt und testet.
- sich auf Namen und Bedeutung möglichst weniger globaler Variablen einigt, die in mehreren Blöcken verwendet werden und so der Kommunikation dienen.
- die Gestaltung der Programmoberfläche (Lage der Knöpfe, …) möglichst in einer Hand lässt.
- die entwickelten Blöcke und Sprites an den Arbeitsplätzen exportiert und dann in ein gemeinsames Programm, dass dann wohl auch die Programmoberfläche gestaltet, importiert.
- überflüssige Sprites, die nur dem Import von Blöcken dienten, wieder löscht.

[15] Demonstriert wird das beim mitgelieferten Tool-Sprite von BYOB.

2.5 Aufgaben

1. Korrigieren Sie die genannten Fehler im Skript „Rechtecke zeichnen“, indem Sie vor dem Zeichnen dafür sorgen, dass die Eckpunkte „richtig“ liegen.

2. Führen Sie Möglichkeiten ein, um
   a: drei Punkte anzuklicken, zwischen denen ein „schönes“ Dreieck gezeichnet wird.
   b: bei jedem Mausklick ein Rechteck mit zufällig gewählter Breite und Höhe zu zeichnen. Benutzen Sie die Kachel pick random <min> to <max>.
   c: eine Variable mit Scrollbar einzuführen, mit der sich nur die Werte 1, 2 und 3 einstellen lassen. Je nach Variablenwert erscheinen bei Mausklick Rechtecke, Dreiecke oder Rauten.

3. Mithilfe der *Zählschleife* repeat <number> lassen sich wie im Beispiel gezeigt Kreise zeichnen. Lassen Sie Kreise unterschiedlicher Größe zu.

4. Komplizierte Figuren lassen sich erstellen, indem dem aktuellen Sprite entsprechende Kostüme verpasst werden. Diese lassen sich auf den Bildschirm drucken, indem das Kostüm ausgewählt wird und danach an der richtigen Stelle ein Abdruck mithilfe der stamp-Kachel hinterlassen wird. Wenden Sie dieses Verfahren zur Erstellung von Zufallsgrafiken an.

5. Benutzen Sie entweder die eingebaute Kamera Ihres Computers oder fertige Bilder, die mithilfe einer Digitalkamera (z. B. vom Handy) erstellt wurden. Importieren Sie Bilder als neue Kostüme. Benutzen Sie diese Bilder dann, um Benutzer auf Fehleingaben hinzuweisen. Zu den Bildern sollten passende Sprechblasen mit geeigneten Texten erscheinen.

6. Führen Sie einen Radiergummi ein, mit dessen Hilfe sich Bildteile wieder löschen lassen.

7. Führen Sie als neue Farbmöglichkeit Farbverläufe ein, die die gezeichneten Figuren durch Farbwechsel im Innenbereich „schön“ ausmalen.

8. Entwickeln Sie eine Schriftart, deren Zeichen vom Stift auf den Bildschirm gemalt werden können. Lassen Sie dann Texte ausgeben.

9. Versehen Sie ein Sprite mit Kostümen, die die einzelnen Zeichen einer Schriftart darstellen. Lassen Sie Texte ausgeben, indem Sie die stamp-Kachel verwenden.

3 Simulation eines Federpendels

Neben der weitgehenden Syntaxfreiheit sind die exzellenten Visualisierungsmöglichkeiten und das gutmütige Verhalten von BYOB bei Fehlern ein Anreiz für die Lernenden, experimentell vorzugehen und so eigene Ideen zu erproben. Neben dem analytischen Top-Down-Vorgehen ergibt sich so ein Bottom-Up-Weg des Trial-and-Error, der für Programmieranfänger wichtig ist, weil sie damit erst einmal Erfahrungen auf diesem Gebiet erwerben können, die dann später selbstverständlich zu systematisieren sind. Experimentelles Vorgehen öffnet so gerade am Anfang Möglichkeiten zum selbstständigen Problemlösen statt zum Nachvollziehen vorgegebener Ergebnisse.

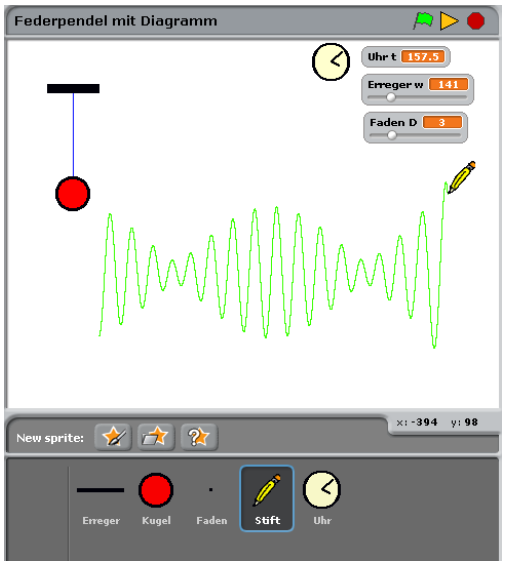
Im Bereich der Simulation, zu der wir auch viele der üblichen Spiele rechnen können, finden wir genügend einfache, aber nicht triviale Problemstellungen, die schon von Anfängern bei etwas gutem Willen bearbeitet werden können. Experimentelles Arbeiten erfordert dabei natürlich ein Interesse, eigene Ideen zu entwickeln. Wir brauchen also Beispiele, die genügend Motivation erzeugen.

Als Beispiel wählen wir die Simulation eines einfachen Federpendels, das an einem periodisch schwingenden Erreger hängt. Ich weiß schon, dass ein Beispiel aus der Physik nicht bei allen Lernenden sehr motivierend wirkt – eher im Gegenteil. Aber ich gebe die Hoffnung nicht auf!

Das Beispiel enthält mehrere weitgehend unabhängig arbeitende Teile, sodass es sich arbeitsteilige Gruppenarbeit geradezu aufdrängt.

Wir identifizieren

- einen Erreger, die schwarze Platte oben-links, der periodisch vertikal schwingt. Seine Frequenz w (statt ω) wird angezeigt und kann mit dem Schieberegler der Variablenanzeige geändert werden.
- eine Kugel, die relativ dumm am Faden hängt, aber immerhin so viel Physik versteht, dass sie die Grundgleichung der Mechanik kennt.
- einen Faden, der sich selbst immer wieder neu zeichnen muss, damit wir keine überstehenden Enden am Bildschirm sehen.
- einen Stift, der das Weg-Zeit-Diagramm der Bewegung aufzeichnet.
- eine Uhr für die gemeinsame Zeit.



der Bildschirmaufbau

die Sprites

3.1 Die Uhr

Wir löschen Alonzo, weil der nur stört, und zeichnen stattdessen eine einfache Uhr. Diese setzt, nachdem sie mithilfe der grünen Flagge gestartet wurde, den in BYOB eingebauten Timer auf Null und übernimmt die aktuelle Zeit fortlaufend in eine Variable t, die den anderen Sprites als Systemzeit zur Verfügung steht.

Da die Zeit t logisch zur Uhr gehört, vereinbaren wir sie als lokale Variable. Angezeigt wird sie damit in der Variables-Kategorie unter einem grauen Strich. (Die globalen Variablen stehen darüber.) Der Zugriff auf lokale Variable erfolgt von anderen Sprites aus über die <attribute> of <sprite> - Kachel. 

Die Uhr greift auf keinerlei externe Größen zu und kann deshalb in allen BYOB-Programmen verwendet werden. Wir exportieren sie als Uhr.ysp.

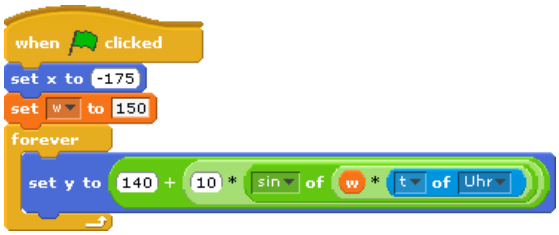
Erweiterung: Lassen Sie die Uhrzeit (Minuten und Sekunden) vom Sprite anzeigen, indem die Zeiger richtig bewegt werden. Sie sollten dazu die Uhr „hohl“ zeichnen!

3.2 Der Erreger

Wir zeichnen ein einfaches Rechteck, das eine irgendwo aufgehängte Platte symbolisiert. Da die Platte nur vertikal schwingen soll, benötigt sie eine feste x-Koordinate am Bildschirm (hier: -175) sowie eine Ruhelage in y-Richtung (hier: 240). Um diese schwingt sie mit einer fest eingestellten Amplitude (hier: 10) mit einer variablen Kreisfrequenz ω (hier: 150). Im Laufe der Zeit t, die anfangs den Wert Null hat, berechnet sich die y-Koordinate dann zu

$$y = 140 + 10 * \sin(\omega t) \; .$$

Diese Angaben lassen sich direkt in ein Skript übersetzen.



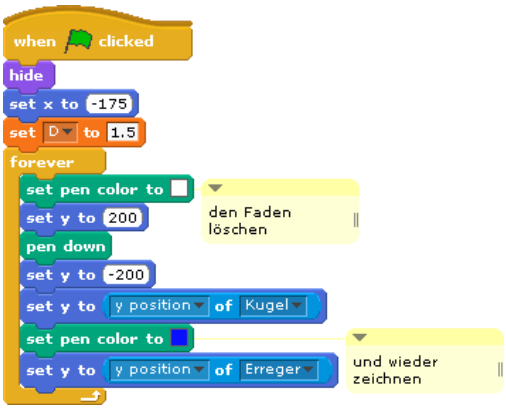
Das Skript beginnt seine Arbeit, wenn die grüne Flagge angeklickt wird. Da die Skripte der anderen Teile zum gleichen Zeitpunkt gestartet werden müssen, bietet sich diese Möglichkeit an.

Interessanter sind die benutzten Variablen. Die Frequenz wird in keinem anderen Skript benötigt und sollte daher lokal vereinbart werden. Die Zeit wird von der Uhr importiert. Wir exportieren das Sprite als Erreger.ysp.

Erweiterung: Lassen Sie auch die „Labordecke“ zeichnen, gegen die der Erreger schwingt. Alternativ dazu kann sich auch eine Welle drehen, die über eine Umlenkrolle zu einer senkrechten periodischen Bewegung führt.

3.3 Der Faden

Der Faden ersetzt die Feder. Er verfügt nur über eine einzige Eigenschaft, die Federkonstante D. Diese wird einmal auf einen festen Wert gesetzt, danach wird eine weiße senkrechte Linie am Ort des Fadens gezeichnet, die seine alte Darstellung löscht (das geht natürlich auch eleganter). Danach wird die momentane Fadenauslenkung gezeichnet. Wir exportieren das Sprite als Faden.ysp.



hier wurden mal Kommentare an das Skript gehängt

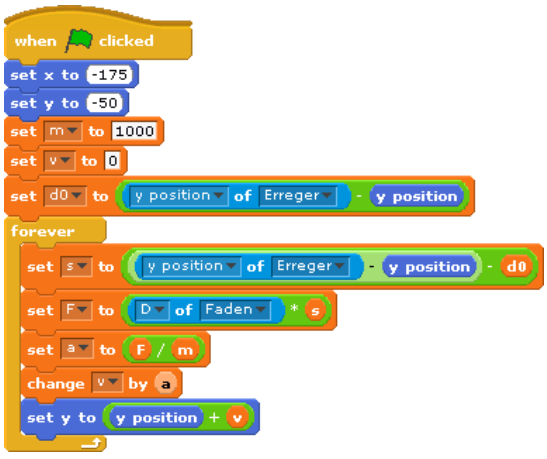
Erweiterung: Zeichnen Sie statt des einfachen Fadens eine Spiralfeder mit einer konstanten Anzahl von Windungen, die sich dehnt und wieder zusammenzieht.

3.4 Die Kugel

In die Kugel werden unsere physikalischen Kenntnisse „eingebaut“, die recht dürftig sein können: Wir kennen die Grundgleichung der Mechanik $F = m*a$ sowie das Hookesche Gesetz $F = D*s$, wobei es sich bei s um die Auslenkung aus der Ruhelage handelt. Weiterhin sind die Beschleunigung a als Geschwindigkeitsänderung pro Zeiteinheit und die Geschwindigkeit v als Wegänderung pro Zeiteinheit bekannt. Sonst nichts.

Als lokale Variable benötigen wir die zu berechnenden Größen sowie die Masse m.

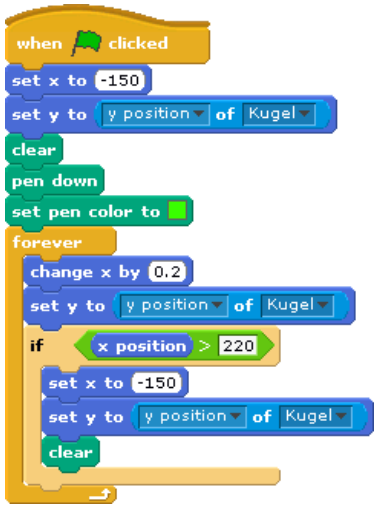
Wir setzen diese Kenntnisse in eine Folge von Befehlen um: wir bestimmen die momentane Auslenkung s, daraus F, daraus a, daraus v und daraus die neue Position. Auch fertig. Wir exportieren das Sprite als Kugel.ysp.



Erweiterung: Führen Sie eine Reibungskonstante R ein, die die Geschwindigkeit um einen bestimmten (kleinen) Prozentsatz mindert. R soll auch interaktiv in einem sinnvollen Bereich änderbar sein.

3.5 Der Stift

Der Stift verfügt über keine zusätzlichen lokalen Variablen. Er wandert langsam von links nach rechts und bewegt sich in y-Richtung zur y-Position der Kugel. Dabei schreibt er. Wir fügen als kleines Schmankerl die Funktion ein, dass er neu zu schreiben beginnt, wenn er den rechten Rand erreicht hat. Wir exportieren das Sprite als Stift.ysp.



Erweiterung: Führen Sie eine Möglichkeit ein, dass der Stift seine x-Position direkt aus der Systemzeit ableitet. Er soll auch mit unterschiedlichen Geschwindigkeiten laufen können.

3.6 Das Zusammenspiel der Komponenten

Vor Beginn der Arbeit müssen die globalen Variablen (hier: keine) und die lokalen Attribute der Sprites (hier: Masse, Federkonstante, …) festgelegt werden. Die einzelnen Gruppen erzeugen dann funktionslose Sprites dieses Namens, die über die genannten Attribute verfügen. Damit können sie in ihren Skripten auf entsprechende Attribute zugreifen. Als Alternative fügen sie einfach sinnvolle konstante Werte ein.

Der Zusammenbau beginnt mit einem leeren Projekt, in das die einzelnen Sprites importiert werden. Danach können die Attributwerte für die jeweils anderen Sprites festgelegt werden (durch das Auswahlmenü von <attribute> of <sprite>) und es wird eingestellt, welche Variablen sichtbar sein sollen. Fertig.

3.7 Weshalb handelt es sich um eine Simulation?

Unser Beispiel enthält zwar ein paar physikalische Grundkenntnisse, aber über Resonanz, Schwebung usw. ist darin nichts zu finden. Wir überprüfen mit dem Programm, ob die *denknotwendigen Folgen* (nach Heinrich Hertz) der Grundkenntnisse mit den Beobachtungen im Experiment übereinstimmen, ob unsere Vorstellungen von den physikalischen Zusammenhängen also das beobachtete Verhalten ergeben. Wir simulieren ein System, um unsere Vorstellungen zu überprüfen.

Etwas ganz anderes ist eine Animation, in die das beobachtete Verhalten einprogrammiert wird. Hier können sich keine neuen Phänomene ergeben, weil alles bekannt ist. *Animationen* stellen etwas dar, *Simulationen* können zu echten Überraschungen führen.

4 Ein Barcodescanner

Als letztes Beispiel für die Nutzung von Blöcken zur Aufteilung eines Problems soll ein einfacher Barcodescanner dienen. Zusätzlich können wir die Einbeziehung externer Hardware in Form eines Picoboards demonstrieren.

4.1 Der EAN-8-Code

Beim EAN-8-Code (European Article Number) handelt es sich um einen einfachen Code, der immer aus genau acht Zeichen besteht. Er dient meist zur Auszeichnung von Artikeln (Waren). Dazu enthält er drei Ziffern, die den Hersteller bezeichnen, und vier Ziffern, die zum Artikel gehören. Zuletzt kommt eine *Prüfziffer*.

Die Ziffern werden durch wechselnde schwarze und weiße Streifen unterschiedlicher Breite codiert, wobei auch die weißen Streifen mitzählen. Die Streifenbreite kann 1, 2, 3 oder 4 betragen – in willkürlichen Einheiten. Eine (hier etwas vereinfachte) Darstellung der Ziffern 0 bis 9 lautet:

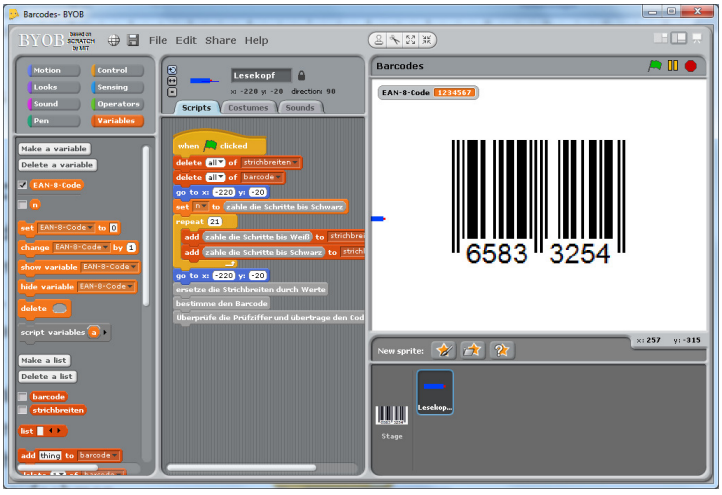
Ziffer	Code	Strichcode
0	3211	
1	2221	
2	2122	
3	1411	
4	1132	
5	1231	
6	1114	
7	1312	
8	1213	
9	3112	

Der Code wird von zwei Doppelstreifen an den Enden und in der Mitte eingerahmt, meist etwas länger gezeichnet (s. o.).

4.2 Blöcke als Strukturierungshilfe

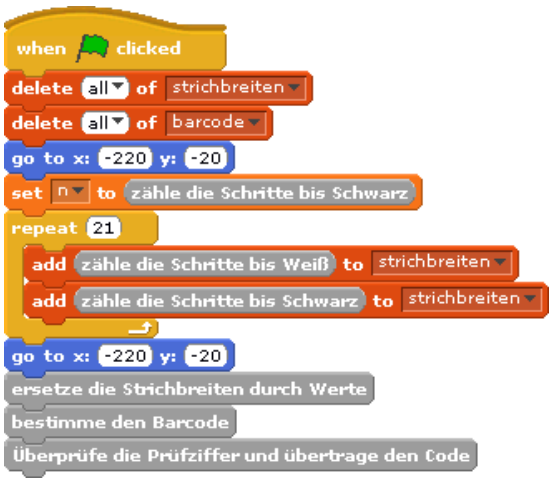
mehrere Kostüme

Wir gehen davon aus, dass die Bühne (*stage*) über unterschiedliche Kostüme verfügt, die jeweils einen EAN-8-Bar-code darstellen.[16] Weiterhin benutzen wir zwei globale Listen strichbreiten und barcode, die die von einem Sprite namens Lesekopf ermittelten Strichbreiten bzw. den ermittelten Code aufnehmen. Der gelesene Code soll in einer Variablen EAN-8-Code dargestellt werden. Damit ergibt sich das dargestellte Szenario. (Der Lesekopf ist das kleine blaue Ding mit dem roten Kopf am linken Rand der Bühne.)



[16] Generatoren für Barcodes finden sich leicht im Internet.

Wir löschen anfangs die Listen, lassen dann den Lesekopf bis zum Barcode laufen. Dann geht es los: wir zählen so oft wie nötig abwechselnd die Schritte bis zur nächsten Markierung und tragen sie in die Liste strichbreiten ein. Danach ersetzen wir die gemessenen Pixelwerte durch die entsprechenden Werte 1 bis 4, bestimmen daraus den Code, prüfen seine Korrektheit und übertragen den Code als Zeichenkette in die dafür vorgesehene Variable. Zur Beschreibung dieses Vorgehens können wir funktionslose leere Blöcke erzeugen, die als Platzhalter im Hauptskript dienen. Wir müssen nur darauf achten, den richtigen Blocktyp auszuwählen:



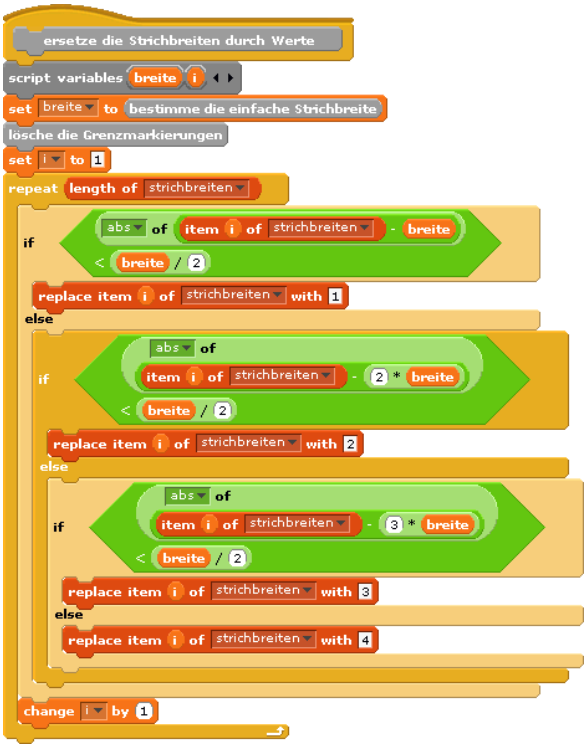
- Bei den ersten Blöcken handelt es sich um Funktionen (*reporter*), die ein Ergebnis zurückgeben. Man erkennt das an der ovalen Form.
- Die letzten drei Blöcke sind Befehle (*commands*), die einfach ausgeführt werden.

Die sehen wir uns jetzt genauer an:

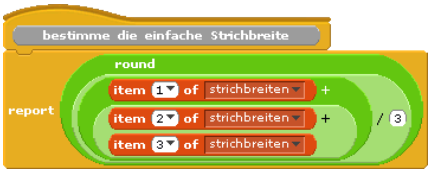
Zähle die Schritte bis Schwarz geht zwei Schritte vorwärts, um ggf. den schon überlaufenen schwarzen Strich zu verlassen, und zählt dann die Schritte bis zum nächsten schwarzen Strich. Diese gibt der Reporter als Ergebnis zurück.

Zähle die Schritte bis Weiß arbeitet entsprechend.

Jetzt müssen wir die im Code versteckten Werte ermitteln. Dazu bestimmen wir zuerst mal die Breite eines einfachen Strichs. Die haben wir ja zum Glück: die ersten drei Striche (2x schwarz, einmal weiß dazwischen) haben genau diese Breite. Dann löschen wir die Grenzmarkierungen und die der Mitte, die brauchen wir nicht mehr. Danach untersuchen wir alle gespeicherten Werte, welcher Wert am besten passt – und geben dabei eine halbe Strichbreite Spiel.



Die einfache Strichreite bestimmen wir also einfach durch Mittelwertbildung der ersten drei Messwerte.



Entsprechend einfach ist das Löschen der Markierungen.

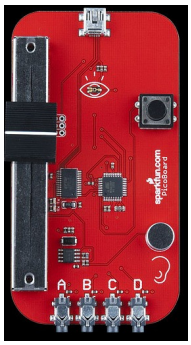
Damit ist das Schlimmste erledigt. Wir können den Barcode bestimmen, indem wir jeweils vier Strichbreiten zu einem Code (als Zeichenkette) zusammensetzen und dessen Wert ermitteln. Die erforderliche Codetabelle verpacken wir in einem Reporter-Block Wert von <code>.

An diesem Punkt der Verarbeitung liegt der Barcode in der dafür vorgesehenen Liste vor. Aber stimmt er überhaupt?

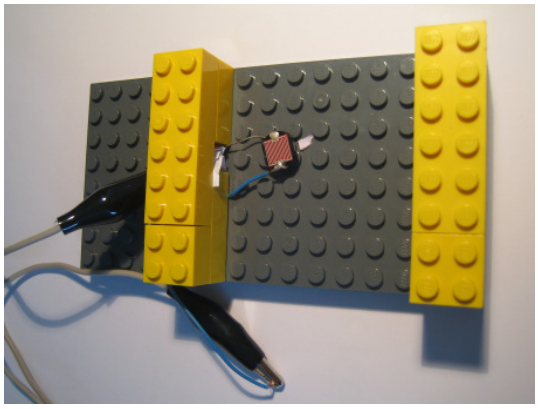
Wir berechnen zuerst die Prüfziffer aus den ersten sieben Ziffern. Dazu werden diese – von hinten beginnend – abwechselnd mit 3 und 1 multipliziert und addiert. Von der Summe wird der Rest bei Division durch 10 genommen und von 10 abgezogen. Sollte sich eine 10 ergeben, dann wird wiederum der Rest genommen.

Stimmt dieses Ergebnis mit der Prüfziffer überein, dann werden die ersten sieben Ziffern angezeigt, sonst ein Fehler.

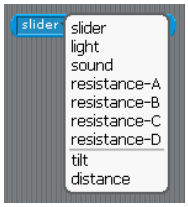
4.3 Die Nutzung des Picoboards



Ein Barcodescanner ist ein Standardbeispiel für die modellhafte Nutzung von externen Geräten zur Erklärung von Teilen der erfahrenen informatisch geprägten Umwelt. Das Board wird einfach – nach Installation des richtigen Treibers – an den USB-Port des Computers angeschlossen. Für unseren Zweck enthält es einen Lichtsensor, der allerdings mitten auf dem Board sitzt und von Schieberegler und Knopf umgeben ist, also also Sensor für einen Scanner nur bedingt taugt. Wir können aber an den vier Eingängen A bis D beliebige Widerstände anschließen, u.a. einfache LDR[17] (hier: LDR03). Diese lassen sich leicht z. B. in LEGO-Bauteile integrieren.



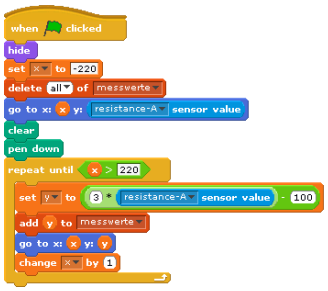
Wir versuchen es mit einer absoluten Primitiv-Version: Der LDR wird einfach auf ein Legoteil gelegt und mit Steinen festgeklemmt. Darüber ziehen wir einen Papierstreifen mit drei unterschiedlich breiten schwarzen Markierungen – möglichst mit gleichmäßiger Geschwindigkeit.

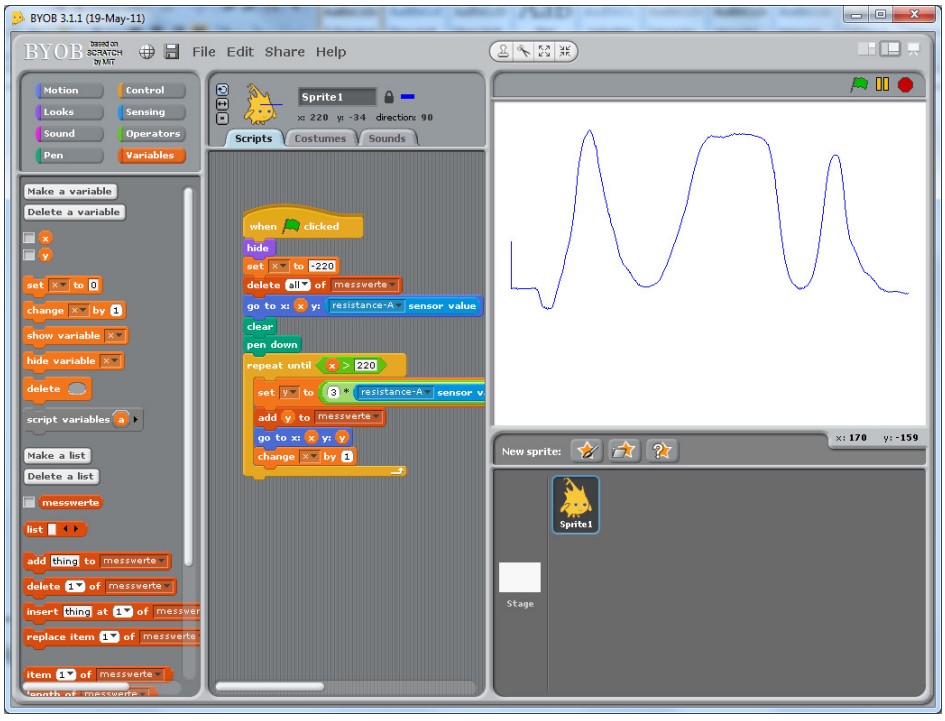


Wir benötigen ein Skript, um die Messwerte aufzunehmen. Einen entsprechenden Block finden wir in der *Sensing*-Rubrik: 

Dort stellen wir den Widerstandswert am Eingang A ein. Dieser wird automatisch auf einen Wert zwischen 0 und 100 skaliert – egal, was wie anschließen.

Die Messwerte stellen wir am Bildschirm dar und schreiben sie zusätzlich in eine Liste messwerte. Ist das geschehen, erfolgt die Auswertung ähnlich wie oben gezeigt.





[17] Light Dependent Resistor

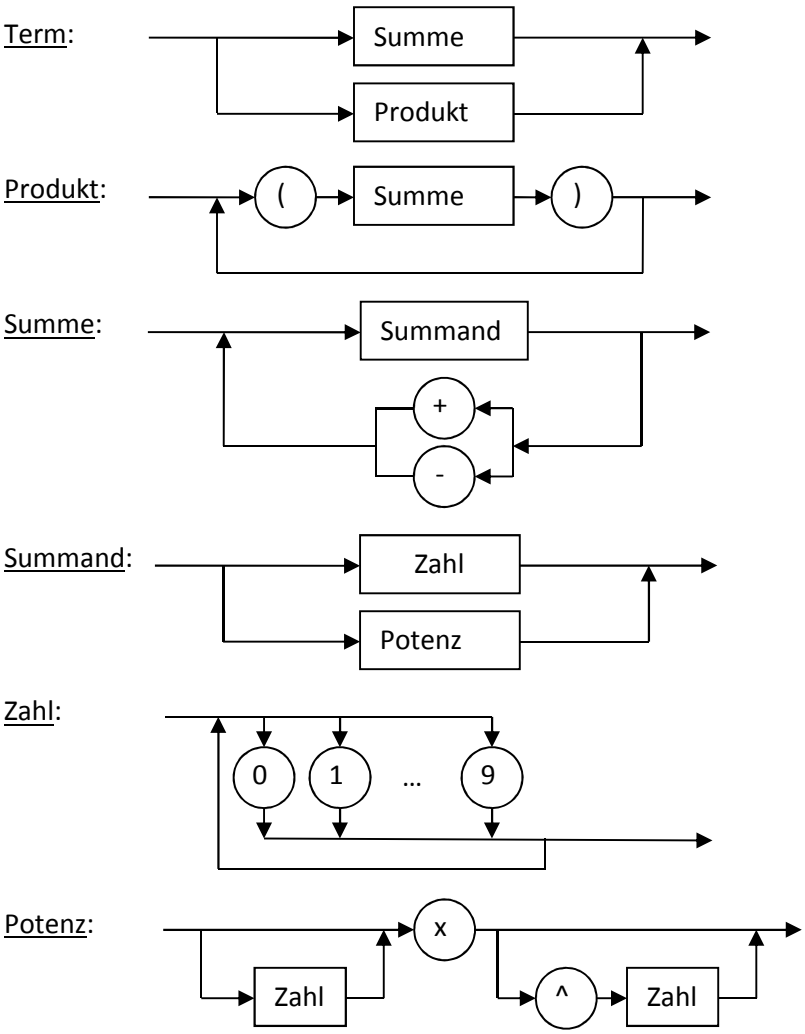
5 Computeralgebra: Blöcke und Rekursionen

Mithilfe des Blockeditors können wir drei Arten von Kacheln (Unterprogrammen) erzeugen: commands (Prozeduren, Methoden), reporter (Funktionen) und predicates (Prädikate). Die Namen können völlig frei gewählt werden und auch aus mehreren Worten bestehen oder Sonderzeichen enthalten. Zwischen diese Worte können Parameter eingefügt werden, wie bei Bedarf auch typisiert werden, um Fehleingaben zu vermeiden. Da in BYOB Skripte auch als Eingaben oder Ausgaben von Blöcken dienen können, ermöglicht BYOB auf einfache Art die Erweiterung der Sprache selbst, also das Erzeugen neuer Blöcke in der *Control*-Kategorie. Die freie Gestaltbarkeit der Blockköpfe gestattet es, die Sprache bei Bedarf bekannten Konventionen anzupassen, z. B. Klammern um die Parameter zu setzen etc. – wenn man das will.

5.1 Funktionsterme

Wir wollen die Möglichkeiten von Blöcken, insbesondere Prädikaten, anhand eines kleinen „*Computeralgebrasystems*“ zeigen. Dazu müssen wir definieren, was wir unter Funktionstermen verstehen:

Syntaxdiagramme

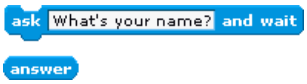


Funktionsterme sind als z. B.: 3 4x (2x-1)(x^2+2) x(x^2)(1-2x^4)

5.2 Funktionsterme parsen

Zur Arbeit mit Funktionstermen brauchen wir natürlich jemanden, der etwas davon versteht. Wir zeichnen deshalb Paul, den kleinen Mathematiker, und machen den danach schlau.

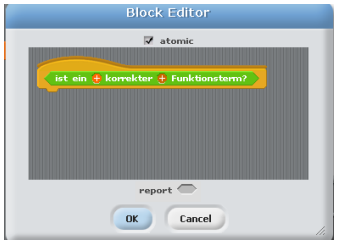
Zuerst einmal muss Paul Funktionsterme einlesen können. Dafür bittet er den Benutzer um eine entsprechende Eingabe mithilfe des Blocks ask <question> and wait aus der *Sensing*-Rubrik. Die Antwort des Benutzers kann anschließend mithilfe des answer-Blocks ermittelt werden. Im Skript weisen wir das Ergebnis einer Variablen zu.

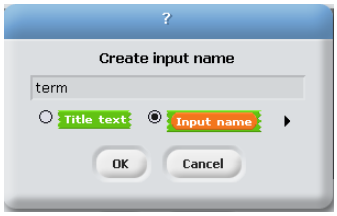


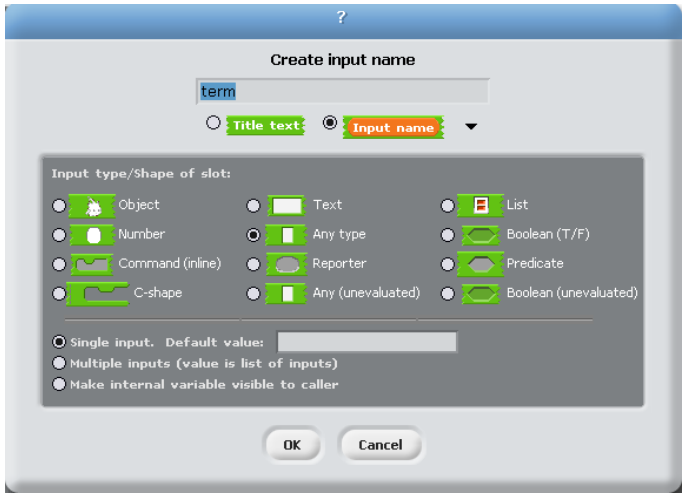
Achtung! BYOB-Fehler

Man darf diesen Reporter nicht direkt ohne Zuweisung aufrufen, weil BYOB sonst zu seltsamen Reaktionen neigt!

Als nächstes überprüfen wir, ob die Eingabe korrekt ist – wir schreiben also ein Prädikat. Nachdem wir einen aussagekräftigen Titel gewählt haben, landen wir im Blockeditor. Dort zeigen wir mit der Maus an die Stelle in der Titelzeile, an der ein Parameter (hier: der eingegebene Term) stehen soll – es erscheinen kleine Kreuze. Klicken wir eines an, dann können wir den Bezeichner des Parameters eingeben. Da es sich hier um eine Zeichenkette handelt, typisieren wir diesen, indem wir auf den kleinen Pfeil rechts davon klicken. Es erscheint ein umfangreicheres Auswahlmenü, in dem wir in den oberen beiden Zeilen sechs Typen (fünf davon mit weiß hinterlegtem Hintergrund) finden, aus denen wir den Typ Text auswählen.



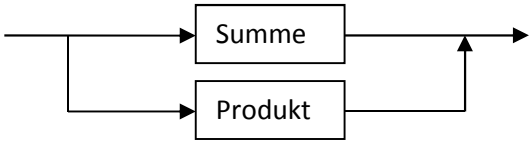




Sobald wir den Kopf eines Blocks definiert haben, steht uns die entsprechende Kachel in der ausgewählten Kategorie zur Verfügung. Damit kann diese – noch ohne Inhalt – sofort weiterverwendet werden – auch im Blockeditor des eigenen Blocks. Damit werden rekursive Definitionen möglich.

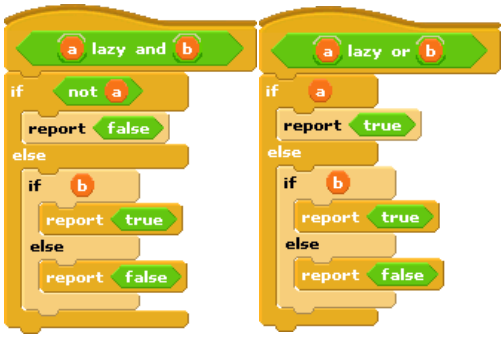
Rekursive Blöcke

Wir nutzen diese Fähigkeit für ein Top-down-Vorgehen aus und definieren zwei (noch) leere Prädikate ist <term> eine Summe? und ist <term> ein Produkt? Diese verwenden wir entsprechend den Syntaxdiagrammen in der Definition des ersten Blocks ist <term> ein korrekter Funktionsterm?



BYOB wertet logische Ausdrücke vollständig aus, was auch nett ist, wenn Nebeneffekte zu berücksichtigen sind. Das erhöht allerdings bei baumartigen Aufrufstrukturen gewaltig die Laufzeit, die im Folgenden sowieso schon grenzwertig ist. Deshalb schreiben wir zuerst zwei Prädikate für die *lazy evaluation* boolescher Ausdrücke: der zweite Ausdruck wird nur ausgewertet, wenn der erste nicht schon das Ergebnis bestimmt.

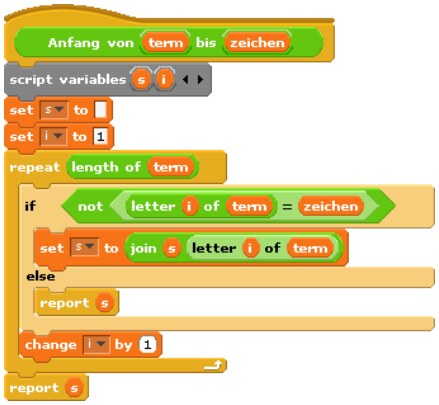
lazy evaluation

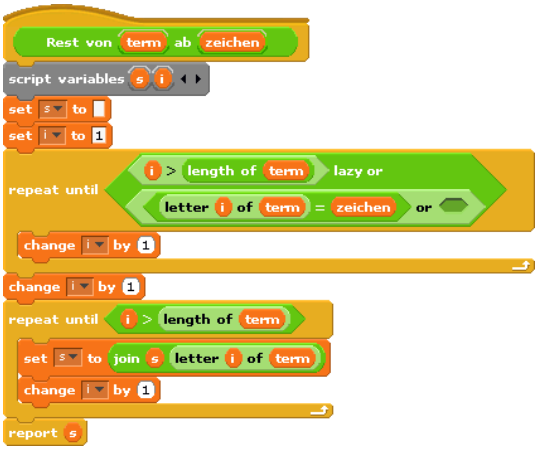


Mit diesen Prädikaten definieren wir jetzt die für die Syntaxanalyse erforderlichen, die sich aus den Syntaxdiagrammen ergeben.

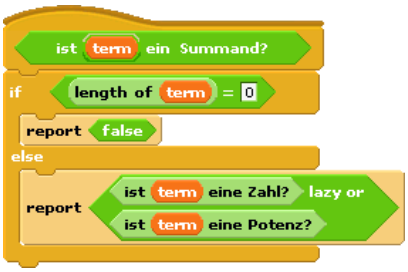
Dieses Verfahren setzen wir für alle Elemente der Sprachdefinition korrekter Funktionsterme fort. Zuerst nehmen wir uns die Summe vor. Diese besteht entweder aus einem einzelnen Summanden oder einem Summanden, gefolgt vom richtigen Operator (+/-) und einer Summe. Das können wir direkt hinschreiben, wenn wir über ein vorerst noch leeres Prädikat ist <term> ein Summand? verfügen. Wie brauchen aber noch etwas mehr. Der eingegebene Term wird ja nicht mehr insgesamt untersucht, sondern wir müssen ihn ggf. in zwei Teile aufspalten: den Anfang von <term> bis <zeichen> und den Rest von <term> ab <zeichen>. Beide Funktionen arbeiten mit Zeichenketten und sind leicht zu schreiben.

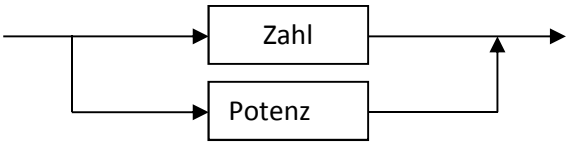
Zeichenkettenverarbeitung

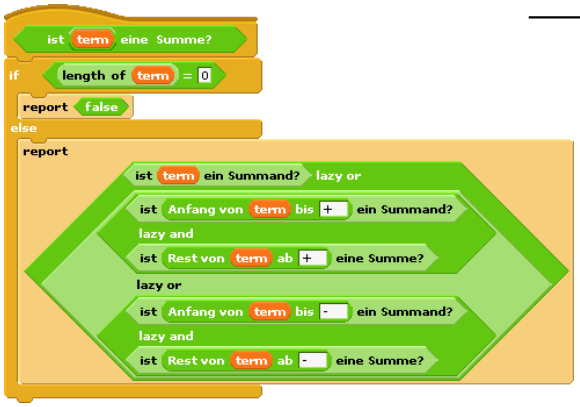


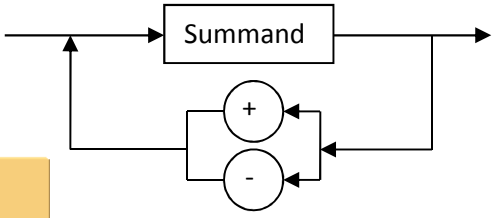


Damit schreiben wir die Prädikate ist <term> ein Summand? und ist <term> eine Summe? – jeweils mit einer zusätzlichen Sicherheitsabfrage.



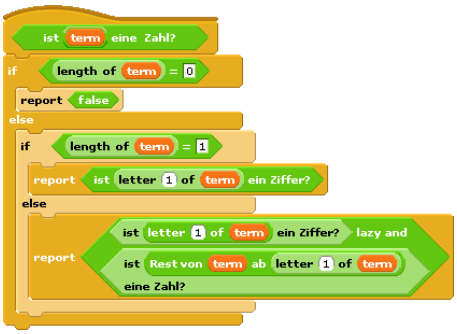


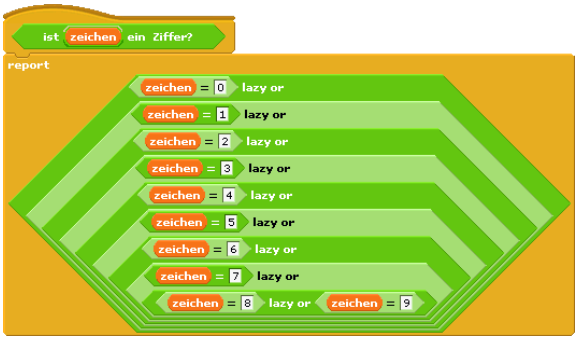




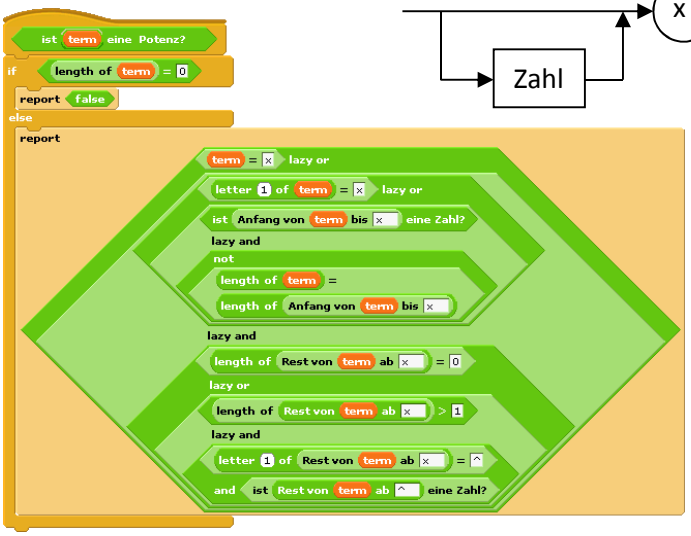
eine rekursive Definition

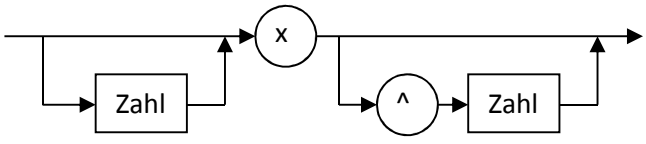
Wir nähern uns dem Ende. ist <term> eine Zahl? ist sehr leicht zu schreiben, wenn man ist <zeichen> eine Ziffer? kennt:



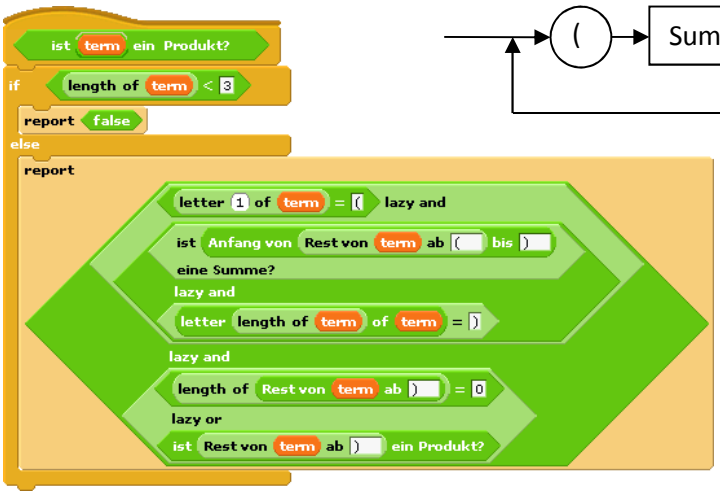


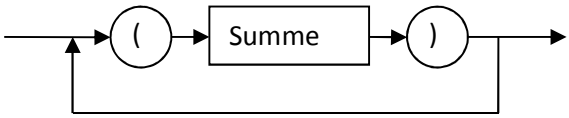
Und wie überprüft man eine Potenz? Das steht ja auch im Syntaxdiagramm – wir müssen nur alle Möglichkeiten abschreiben.





Jetzt fehlt nur noch das Produkt, das sich in direkter Analogie zur Summe formulieren lässt, denn ein Produkt besteht (bei uns) entweder aus einer geklammerten Summe oder einer solchen, gefolgt von einem Produkt.





auch schön rekursiv

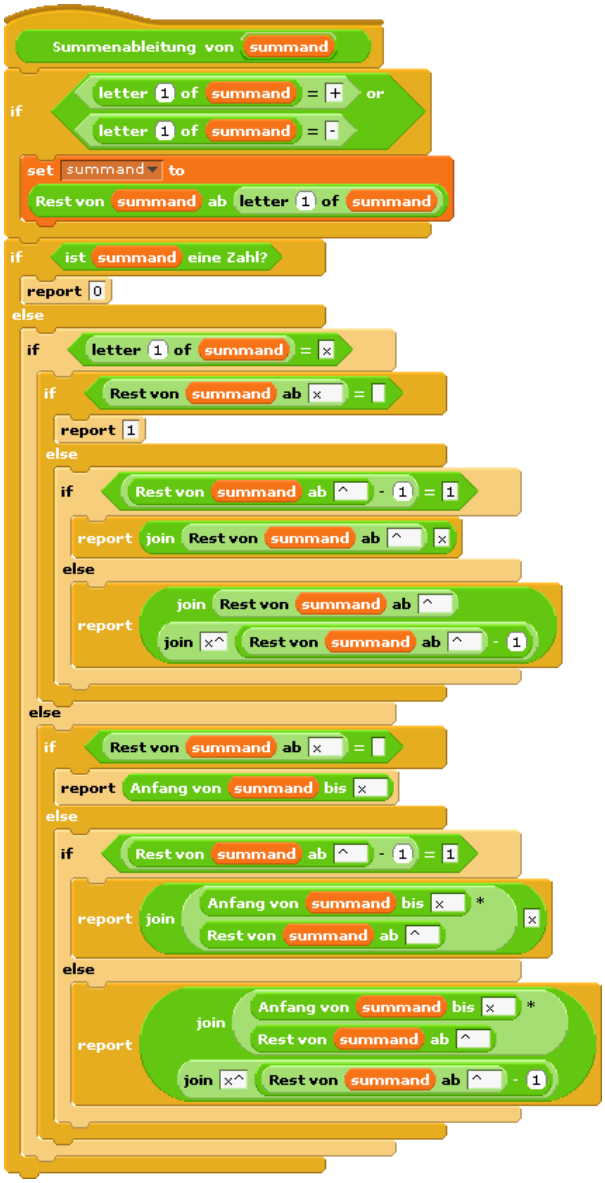
Wir können damit überprüfen (parsen), ob ein eingegebener Term der gewählten Syntax entspricht. Allerdings sind bei längeren Termen die Laufzeiten erheblich. Ziehen wir das Projekt einfach auf Snap!, dann wird es importiert und die Laufzeit ist (in der derzeitigen Version) um der Faktor 15 besser.

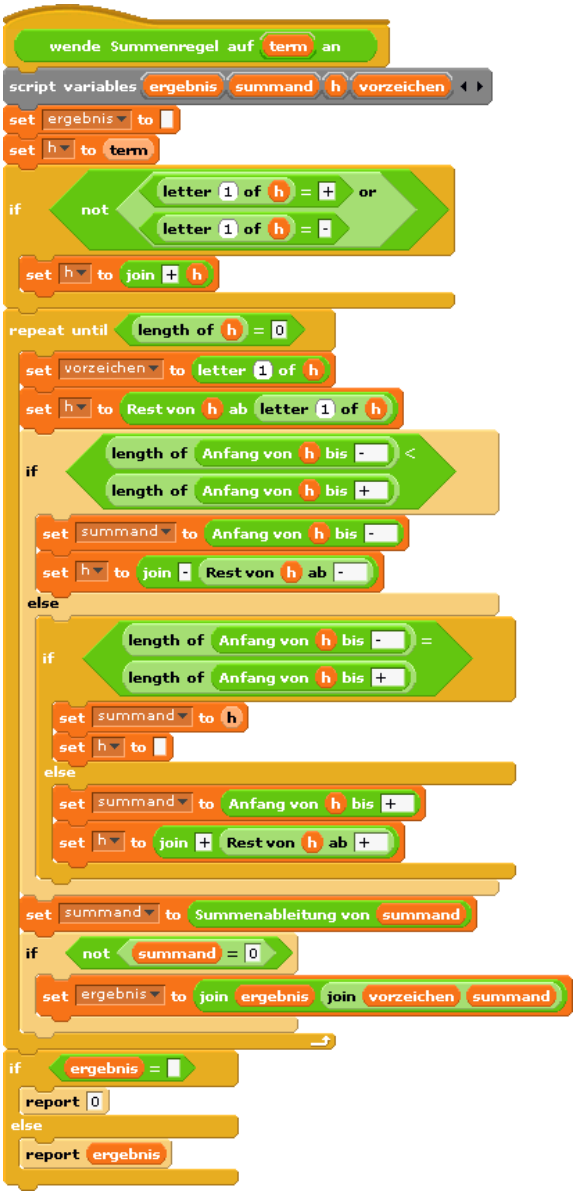
Snap! ist viel schneller

5.3 Funktionsterme ableiten

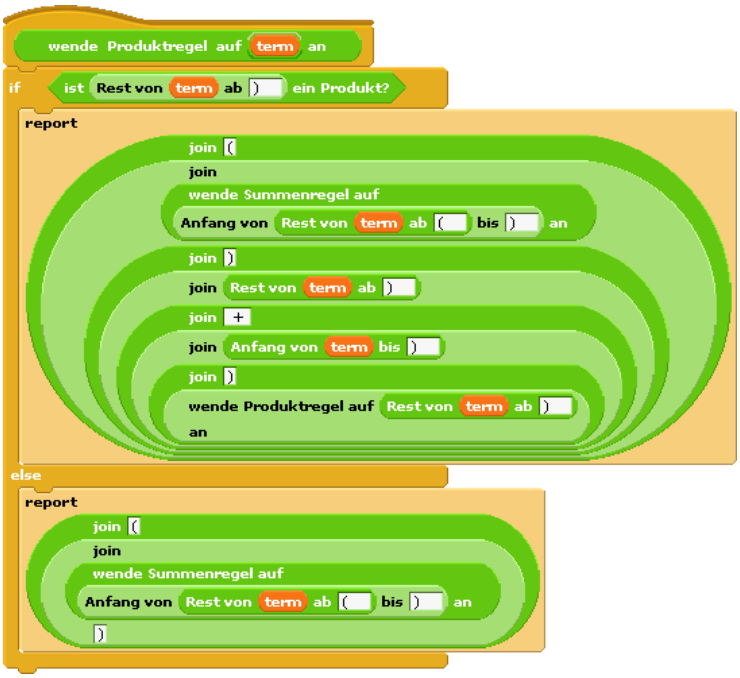
Wir wollen jetzt die erste Ableitung korrekter Funktionsterme bestimmen. Da es bei diesen nur zwei Möglichkeiten für den inneren Aufbau gibt, ist der erste Ansatz einfach:

Bei Anwendung der Summenregel müssen wir die Summanden bestimmen und diese ableiten. Weil wir Zahlen ohne Vorzeichen definiert haben, behandeln wir dieses jeweils gesondert, d. h. wir fügen bei Bedarf ein „+“ hinzu und spalten anschließend das Vorzeichen wieder ab. Anschließend werden die verschiedenen Möglichkeiten entsprechend den Regeln der Mathematik behandelt.



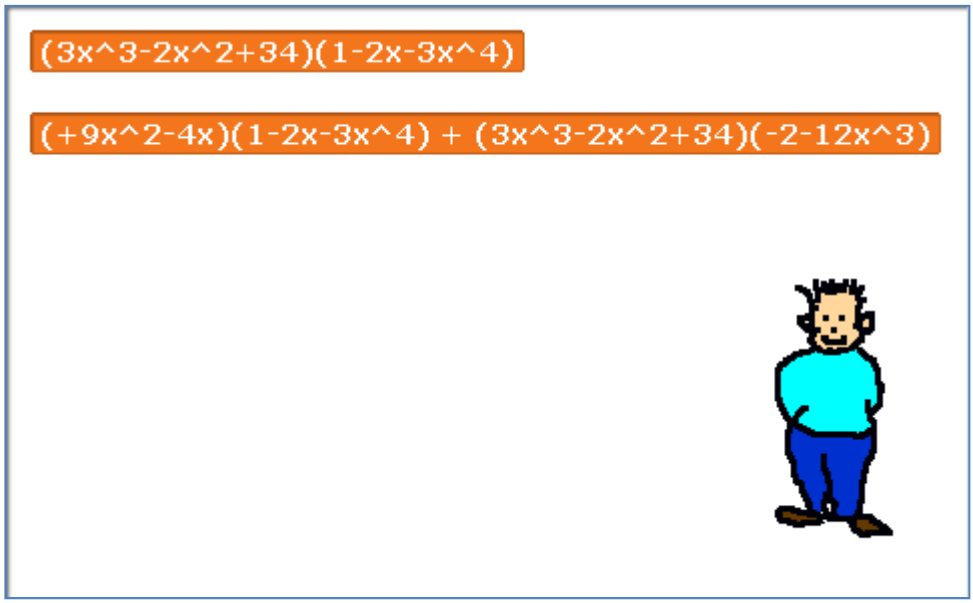


Jetzt fehlt nur noch die Produktregel. Die können wir einfach hinschreiben – unter Hinzufügung einiger Klammern.



Das sieht furchtbar aus, ist aber nur eine Verkettung von Zeichenketten. In Snap! ist es viel kürzer.

Das Ergebnis kann man sogar halbwegs lesen:



5.4 Aufgaben

1. Gestalten Sie die Ausgaben etwas lesefreundlicher: führende „+“ sollen entfernt werden etc.

2. a: Definieren Sie Zahlen mit Vorzeichen und ändern Sie die Verarbeitung der Terme entsprechend ab.
   b: Gehen Sie entsprechend für Gleitpuntzahlen (Zahlen mit Nachkommateil) vor.

3. a: Definieren Sie erweiterte Funktionsterme, die auch Quotienten enthalten können, über Syntaxdiagramme.
   b: Ermöglichen Sie das Parsen dieser Funktionsterme, indem Sie entsprechende Prädikate schreiben.
   c: Bilden Sie Ableitungen, indem Sie auch die Quotientenregel als Zeichenkettenoperation implementieren.

4. Gehen Sie entsprechend der Aufgabe 3 für trigonometrische Funktionen vor.

5. Lassen Sie Funktionsterme zu, die die Anwendung der Kettenregel erfordern. Implementieren Sie entsprechende Prädikate und Zeichenkettenfunktionen.

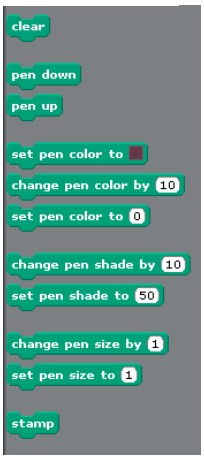
6. a: Lassen Sie die Graphen von Funktionstermen zeichnen, nachdem sie geparst wurden.
   b: Lassen Sie zusätzlich die erste und zweite Ableitung der Funktion zeichnen.

7. Führen Sie einen „Funktions-Taschenrechner“ ein: zuerst wird ein Funktionsterm eingegeben. Ist dieser korrekt, dann können wiederholt Werte eingegeben werden, für die die zugehörigen Funktionswerte ermittelt werden.

6 Rekursive Kurven

Nachdem wir nun die Rekursionsmöglichkeiten von BYOB kennengelernt haben, wenden wir sie auf die implementierte Turtlegrafik an. Die dafür erforderlichen Blöcke verteilen sich auf die Rubriken Pen und Motion.

Rubrik Pen



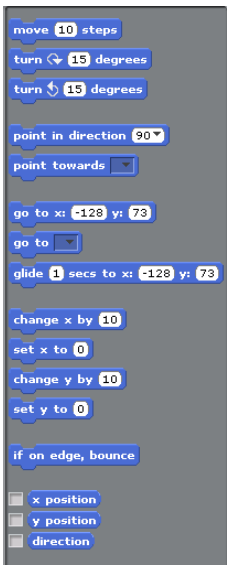
6.1 Die Schneeflockenkurve

Sie entsteht, indem auf einer Seite immer wieder in der Mitte ein Dreieck „ausgestülpt“ wird, solange bis die Seite zu kurz für diesen Prozess ist. In diesem Fall wird die Seite nur als gerade Linie gezeichnet.

zeichne Schneeflockenseite der Länge n

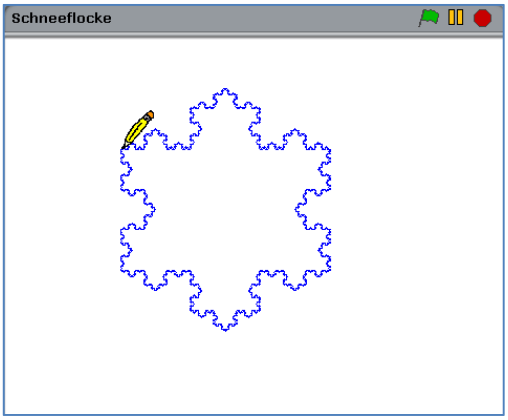
wahr	n < 2
zeichne eine Linie der Länge n	zeichne Schneeflockenseite der Länge n/3
	drehe dich um -60°
	zeichne Schneeflockenseite der Länge n/3
	drehe dich um 120°
	zeichne Schneeflockenseite der Länge n/3
	drehe dich um -60°
	zeichne Schneeflockenseite der Länge n/3

Rubrik Motion



Eine Schneeflocke entsteht, indem ein gleichseitiges „Dreieck“ aus drei solchen Seiten zusammengesetzt wird.

Das Verfahren lässt sich direkt nach BYOB übersetzen:

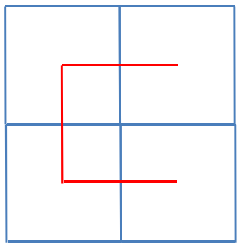


6.2 Die Hilbertkurve

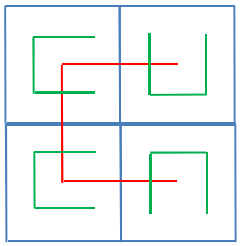
Wir verwenden zur Konstruktion der Kurve eine Version nach László Böszörményi[18].

Die Hilbertkurve ist eine der flächenfüllenden Kurven, die als Generator eine Art Kasten hat. Die Ecken des Kastens liegen in den Mittelpunkten der vier Quadranten eines Quadrats. In der nächsten Stufe wird dieser Kasten um die Hälfte verkleinert, und davon werden vier Versionen in gespiegelter bzw. gedrehter Version in den Quadranten neu angeordnet. Zuletzt werden die kleineren Kästen wie gezeigt miteinander verbunden.

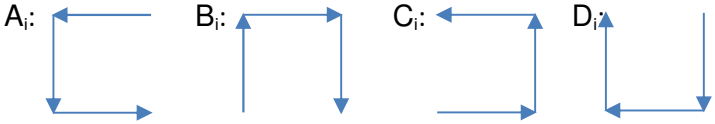
der Generator



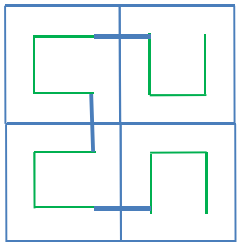
seine Lage im Quadrat



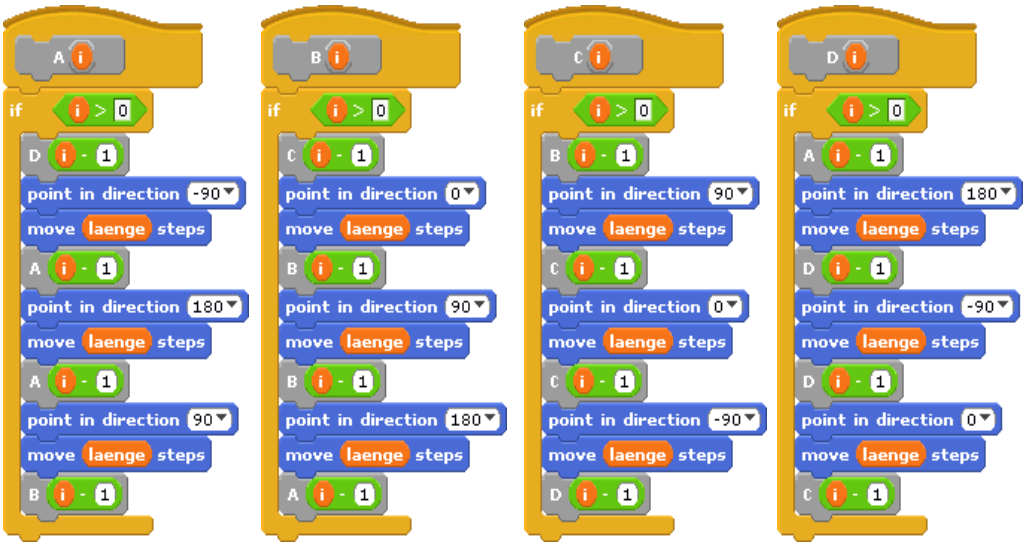
In der Version von Böszörményi werden die Kästen je nach Orientierung und Umlaufrichtung mit A bis D gekennzeichnet.



Aus diesen Elementen wird die Hilbertkurve zusammengesetzt, indem man mit A beginnt und die anderen Elemente „verdreht“ aufruft. Der Parameter i gibt die Rekursionstiefe und damit die Größe der Elemente an. Er wird „heruntergezählt“ bis auf Null.

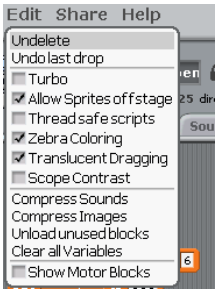


die verkleinerten Kopien und ihre Verbindungen



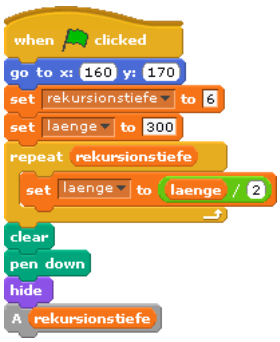
über den Bildrand hinaus zeichnen

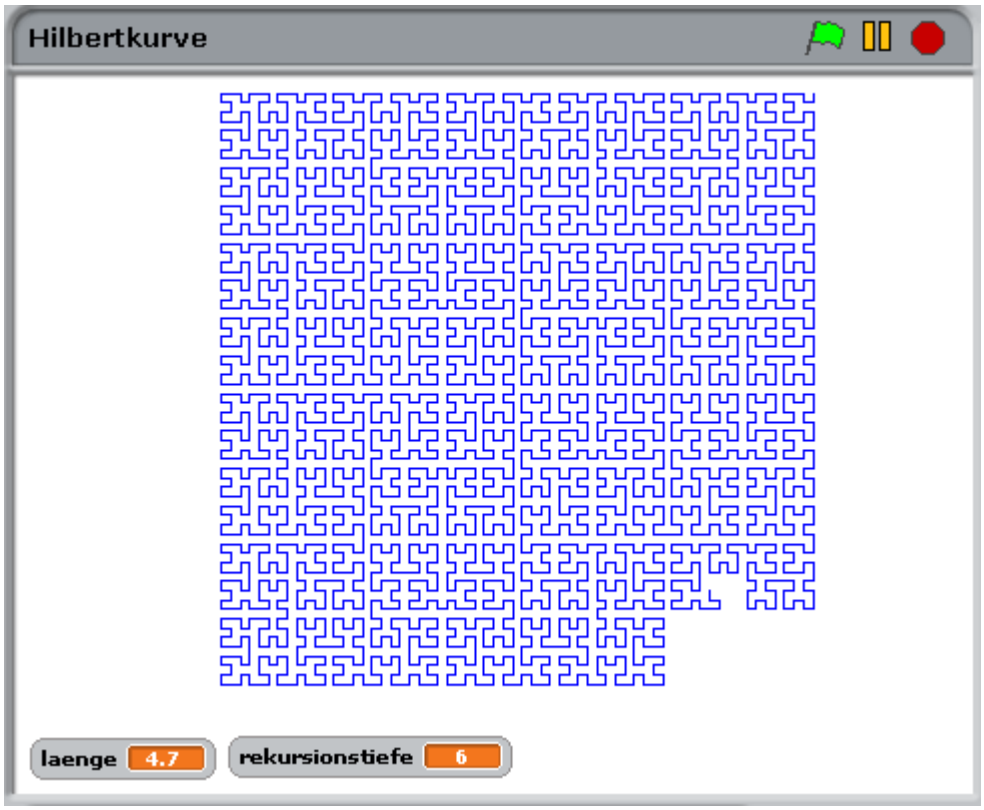
Hinweis: Wenn das Sprite beim Zeichnen an den Rand des Bildbereichs kommt, kann es diesen nicht verlassen und die Kurve wird „verzerrt“. Markieren Sie deshalb im Edit-Menü den Punkt „*Allow Sprites off stage*“.



[18] http://bscwpub-itec.uni-klu.ac.at/pub/bscw.cgi/d11952/10.%20Rekursive%20Algorithmen.pdf

Der Aufruf erfolgt wie beschrieben, nachdem das Sprite zum Anfangspunkt rechts-oben geschickt. Die endgültige Länge der zu zeichnenden Teilstrecken wird aus der Rekursionstiefe ermittelt – und dann wird gezeichnet.





6.3 Aufgaben

1. a: Informieren Sie sich im Internet zum Thema C-Kurve.
   b: Probieren Sie einige Schritte zur Konstruktion der Kurve „per Hand“ aus.
   c: Implementieren Sie ein Skript zum Zeichnen der Kurve in BYOB.

2. Verfahren Sie entsprechend für die Dragon-Kurve.

3. Verfahren Sie entsprechend für die Peano-Kurve.

4. Verfahren Sie entsprechend für die Sierpinski-Kurve.

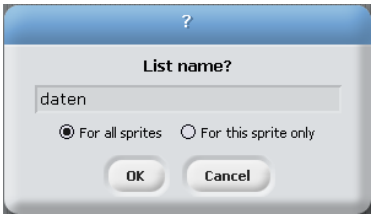
7 Listen und verwandte Strukturen

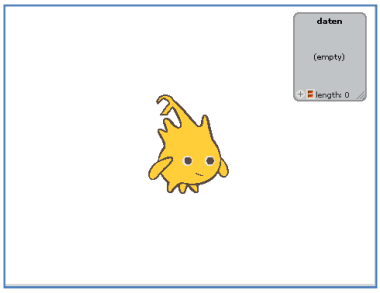
BYOB enthält neben *Variablen*, die alle Daten aufnehmen können, deren Typ also erst zur Laufzeit jeweils evaluiert wird, als grundlegende Datenstruktur *Listen*, aus denen sich alle höheren Strukturen leicht aufbauen lassen. Die Verwendung dieser Strukturen wird zuerst an einem einfachen Fall – dem Sortieren - gezeigt, danach werden Adjazenzlisten für die Wegsuche benutzt und zuletzt werden Matrizen eingeführt – mit der dafür üblichen Kontrollstruktur.

7.1 Sortieren mit Listen – durch Auswahl

die Listen-Rubrik

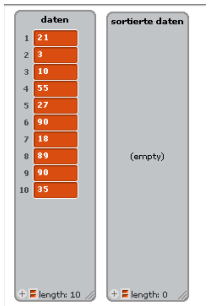
Zuerst einmal benötigen wir eine Liste. Die Blöcke dafür finden wir im unteren Abschnitt der Variables-Rubrik. Zuerst betätigen wir den Make a list-Knopf und erhalten darauf die Möglichkeit, den Namen der Liste festzulegen (hier: daten) und anzugeben, ob die Liste global (für alle Sprites) oder lokal (nur für das aktive Sprite) vereinbart sein soll. Die noch leere Liste wird daraufhin im Bildbereich angezeigt. Bei Bedarf können wir jetzt Daten „per Hand“ durch Betätigung des +-Knopfes in der Anzeige hinzufügen oder wieder löschen.



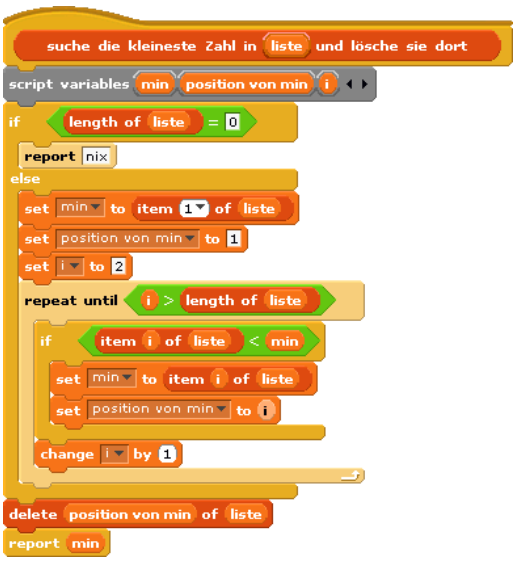


Auf die gleiche Art erzeugen wir eine zweite Liste sortierte Daten, die später die sortierten Daten aufnehmen soll. Zuerst einmal brauchen wir unsortierte – wie üblich Zufallszahlen. Die erzeugen wir mit einem kleinen Skript, wobei wir die Anzahl der Zahlen sowie die zu füllende Liste als Parameter angeben und wie üblich typisieren – als Zahl bzw. Liste. Dabei sollen zuerst die alten Werte in der Liste gelöscht werden.

Rufen wir dieses Skript mit den Parametern 10 und daten auf, dann erhalten wir (bei etwas vergrößerter Anzeige der Listen) das nebenstehende Bild.



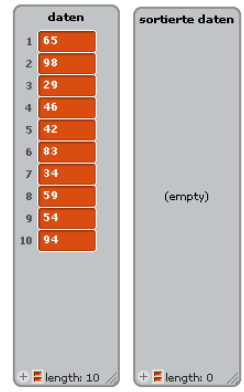
Um diese Daten zu sortieren, wählen wir die jeweils kleinste Zahl aus der Liste der unsortierten Zahlen aus und löschen sie dort. Dazu erzeugen wir drei für den Block lokale Variable min, position von min und i (als Zählvariable). Zur Sicherheit suchen wir nur dann das Minimum, wenn wirklich Zahlen vorhanden sind.[19]



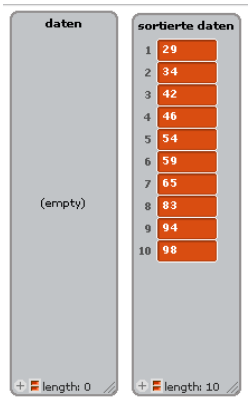
Damit sind wir fast fertig. Das Sortieren wird durchgeführt, indem wiederholt die kleinste Zahl aus der unsortierten Liste in die mit den sortieren Elementen überführt wird.

Die Nutzung ist einfach.

führt zu



führt zu



[19] Falls wir auf nicht vorhandenen Listenpositionen zugreifen, zickt BYOB auch nicht rum. Es liefert einfach ein leeres Element.

7.2 Sortieren mit Listen – Quicksort

Als zweites, rekursives, Beispiel wollen wir Quicksort in der gleichen Umgebung wie oben realisieren.[20] Als Pivot-Element wählen wir einfach das mittlere der jeweilige Teilliste.

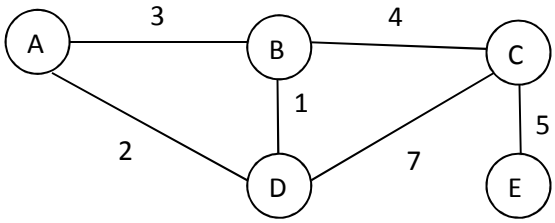
Quicksort wird gestartet, indem die zu sortierende Liste angegeben wird:

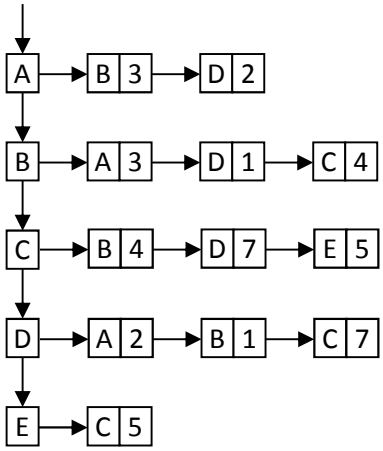
Die eigentliche Arbeit erfolgt im Block teile und ordne die Liste <liste> zwischen <links> und <rechts>.

[20] Das Verfahren findet sich in diversen Versionen im Internet, z. B. unter http://de.wikipedia.org/wiki/Quicksort. Hier wurde eine In-place-implementierung gewählt.

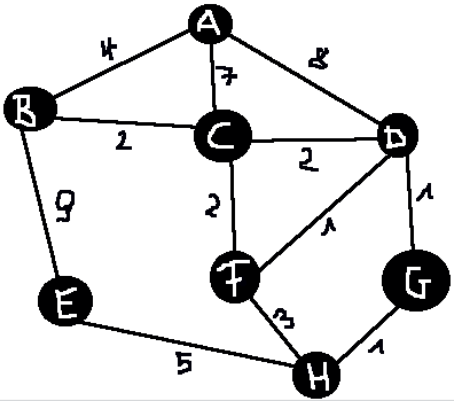
7.3 Kürzeste Wege mit dem Dijstra-Verfahren

Gegeben sei ein Graph durch eine Adjazenzliste. In dieser sind alle Knoten des Graphen aufgeführt, von denen jeweils Listen „abgehen“, in die die Nachbarknoten mit den jeweiligen Entfernungen eingetragen sind: also diejenigen Knoten, zu denen eine direkte Verbindung existiert. Als Beispiele werden ein sehr einfacher Graph und seine Adjazenzliste angegeben.

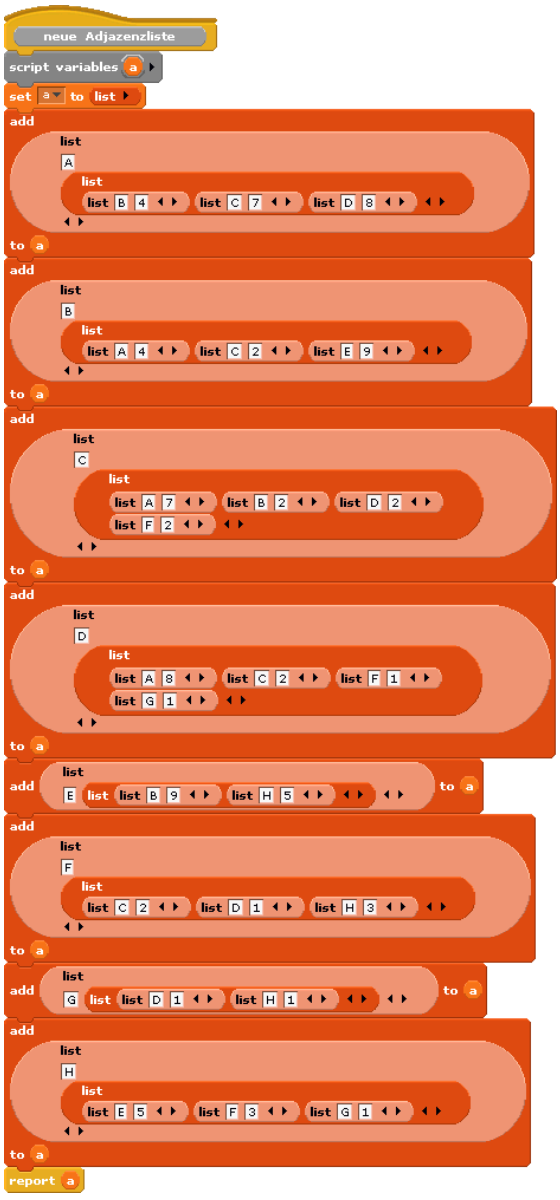




Zur Bearbeitung des Problems benötigen wir natürlich einen Spezialisten: wir löschen Alonzo und zeichnen Mr.D. Dieser muss in der Lage sein, die Adjazenzliste eines gegebenen Graphen zu erzeugen. Den Graphen zeichnen wir einfach auf den Hintergrund – hier sehr geschmackvoll geschehen, und die Liste erzeugen wir statisch durch Einfügen der entsprechenden Elemente in eine lokale Liste, die wir als Ergebnis der Operation zurückgeben.



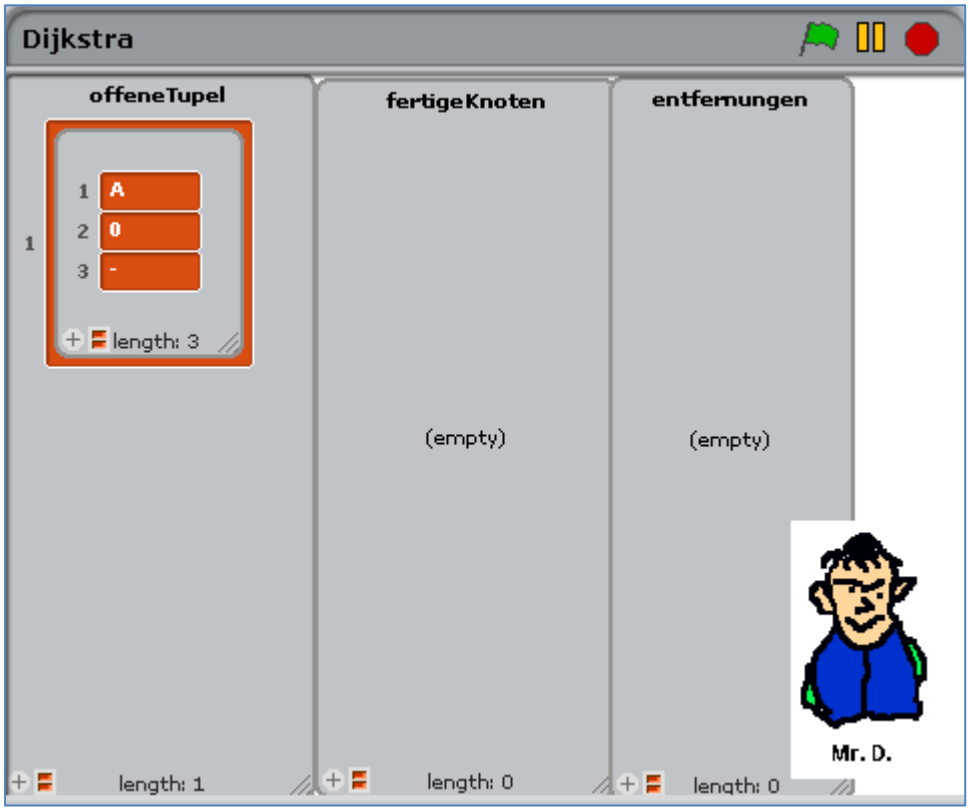
Die Variable adjazenzliste erhält dann diese Werte über eine einfache Zuweisung:



Zur weiteren Bearbeitung benötigen wir noch drei andere Listen:

- Die Liste der offeneTupel nimmt Tupel auf, die den Namen des Knotens, seine Gesamtentfernung vom Startknoten und den Namen des Vorgängerknotens enthalten.
- Die Liste entfernungen nimmt Tupel auf, die den Namen des Knotens und seine Gesamtentfernung vom Startknoten enthalten. Die Liste wird bei Neueintragungen jeweils neu sortiert, sodass der Knoten mit der kürzesten Entfernung vom Start vorne steht.
- Die Liste fertigeKnoten enthält die Namen der Knoten, die bereits fertig bearbeitet wurden.

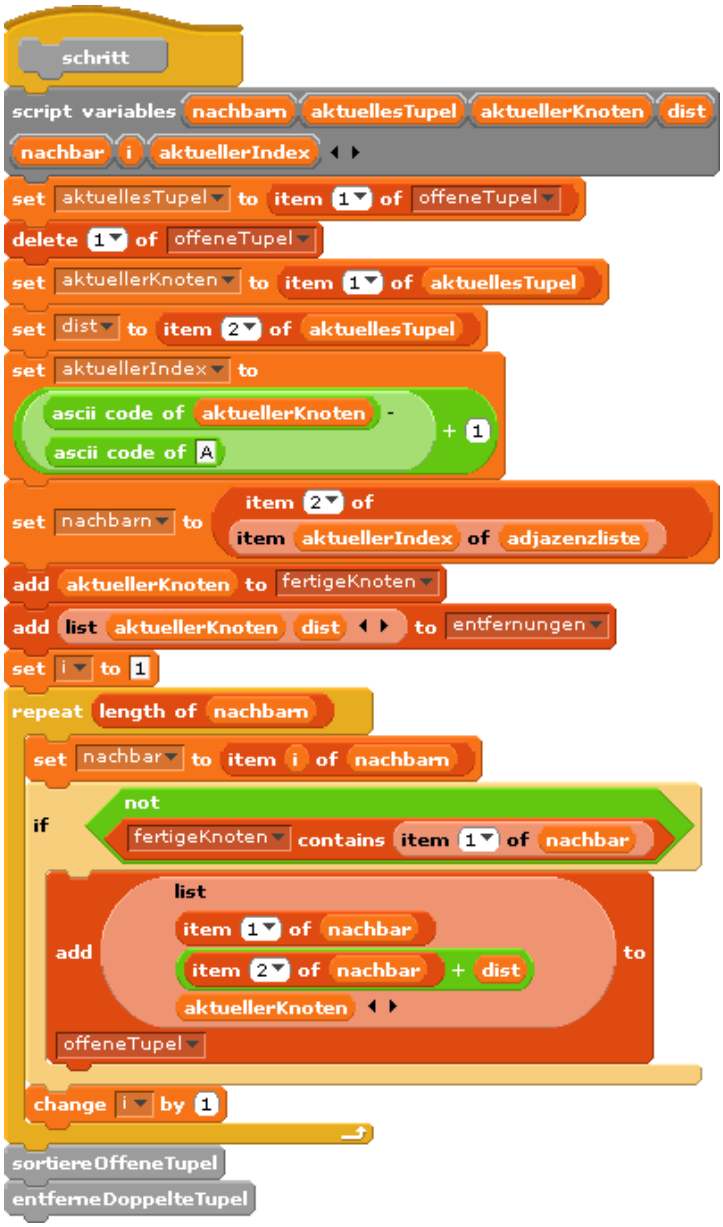
Die Einrichtung dieser Listen für den Start fassen wir in einer Methode vorbereitung zusammen, der auch der Name des Startknotens übergeben wird. Nach ihrem Aufruf ergibt sich das folgende Bild.



Der Zustand zu Beginn der Suche.

Die Wegsuche ist in dieser Version sehr einfach, da der größte Teil der „Intelligenz“ in den Umgang mit den Listen gesteckt wurde.

Die eigentliche Verarbeitung erfolgt damit in der Methode schritt:



lokale Variable vereinbaren

das Tupel mit der bisher kleinsten Entfernung bearbeiten

Knotennamen und Entfernung auslesen und Index in der Adjazenzliste aus dem Namen berechnen

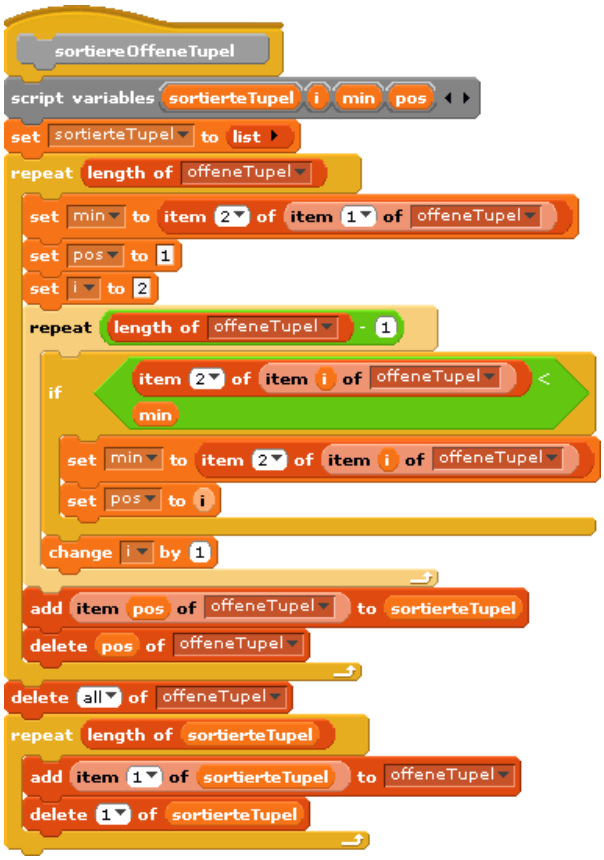
die Nachbarn des Knotens auslesen

Knoten als bearbeitet markieren und Entfernung zum Startknoten merken

alle unbearbeiteten Nachbarn mit neuer Gesamtentfernung und Vorgängerknoten in offeneTupel eintragen

offeneTupel nach der Entfernung sortieren und eventuelle Tupel mit größeren Entfernungen löschen

Wie man sortiert, haben wir gerade weiter oben gesehen. Hier geschieht es durch Auswahl des Kleinsten.



die Liste sortierteTupel nimmt die sortierten Tupel auf

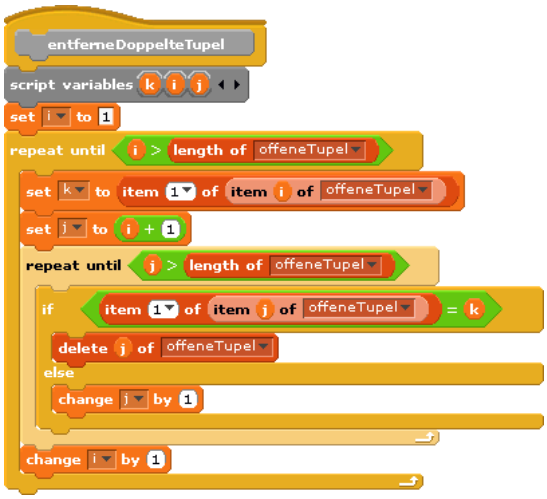
Annahme, dass die kleinste Entfernung ganz vorne steht.

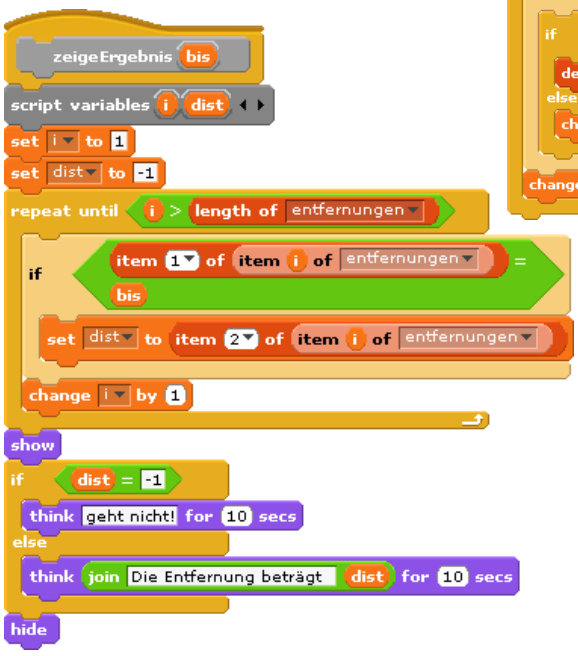
ggf. noch kleinere Entfernungen finden

das Tupel mit der kleinsten Entfernung zu sortierteTupel hinzufügen und in offeneTupel löschen

zuletzt die sortierte Liste zurück kopieren

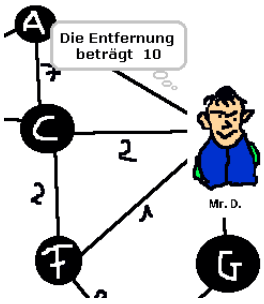
Jetzt steht für jeden Knoten das Tupel mit der kleinsten Entfernung vorne in der Liste. Falls noch andere Tupel für diesen Knoten auftreten, werden sie gelöscht.





Zuletzt müssen wir nur noch die Entfernung zum gesuchten Knoten aus der Liste entfernungen heraussuchen und von Mr.D. anzeigen lassen.

Mr.D. kriegt es raus!



7.4 Matrizen und neue Kontrollstrukturen

Wenn wir Listen haben mit direktem Zugriff auf jedes Element, dann benötigen wir eigentlich keine speziellen Reihungen, Stapel, Schlangen usw. Alle höheren Datenstrukturen lassen sich aus Listen aufbauen. Wir bauen uns trotzdem die Datenstruktur matrix, weil sie traditionell z. B. bei den Adjazenzmatrizen Verwendung findet. Dazu passend schreiben wir uns eine Zählschleife als Kontrollstruktur, um *Matrizen* einfach zu durchlaufen.

Wir verpacken eine Matrix natürlich in einer Liste. Damit wir bei Zugriffen keine Fehler machen, vereinbaren wir (willkürlich) die folgende Listenstruktur:

[[Liste mit Größe der Indexbereiche][Liste mit Daten……]]

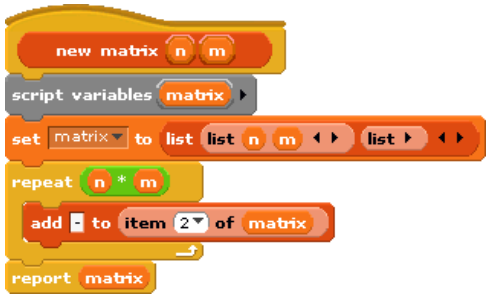
Die Dimension der Matrix ergibt sich dann direkt aus der Länge der ersten Teilliste.

Als Beispiel wählen wir eine zweidimensionale Reihung mit jeweils zwei Werten pro Zeile. Sie hätte die folgende Struktur: [[2,2][1,2,3,4]]

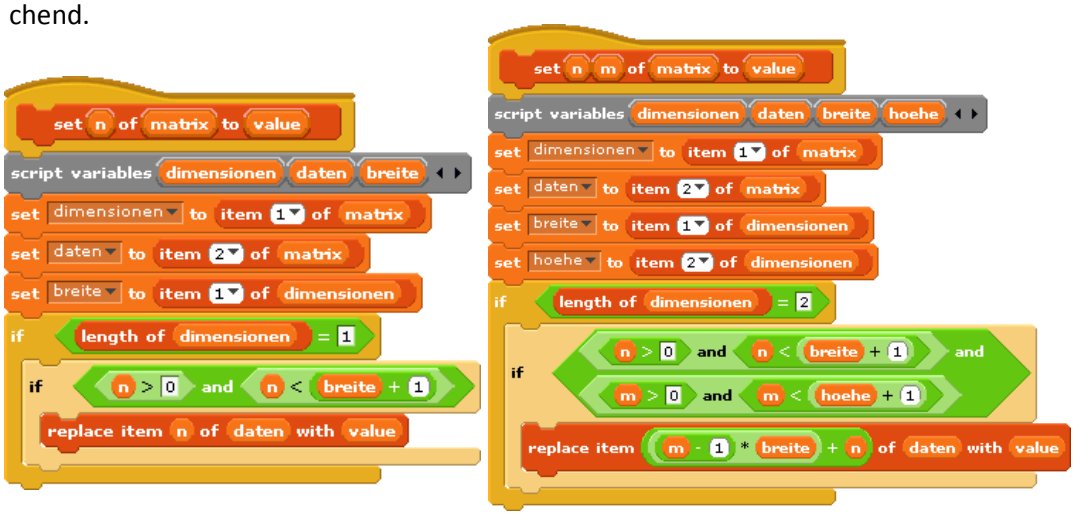
Zuerst einmal legen wir eine eindimensionale Matrix der Größe n an, indem wir eine Liste mit zwei Teillisten erzeugen, von denen die erste die Größe der Matrix (hier: n) enthält und die zweite anfangs leer ist. In diese fügen wir anschließend n-mal ein Minuszeichen (oder sonstwas) ein. Das Ergebnis geben wir zurück.

Die Syntax kann völlig frei gewählt werden, zum Beispiel auch mit eckigen Klammern, wenn man das mag!

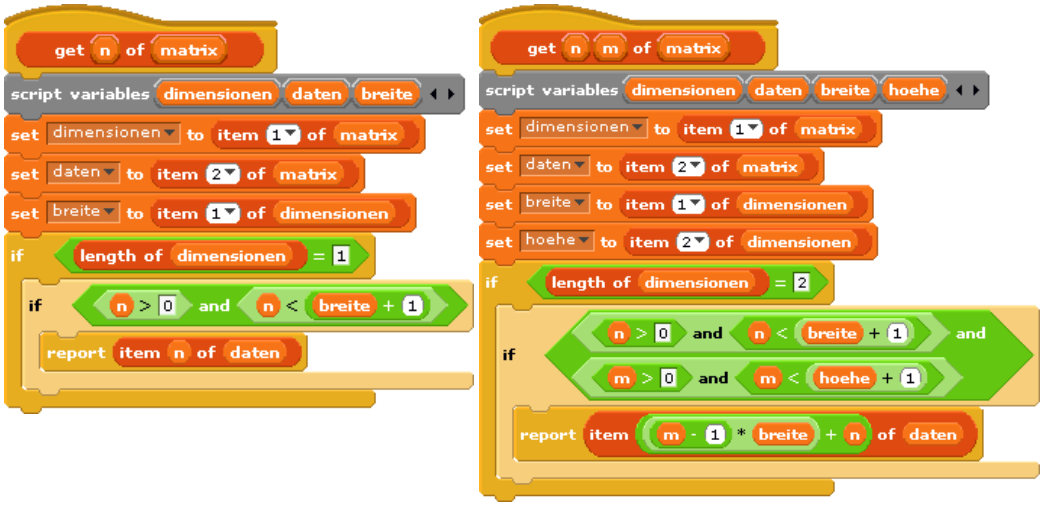
Die Methode zur Erzeugung einer zweidimensionale Matrix unterscheidet sich von der für eine eindimensionale durch ihre *Signatur*: sie enthält zwei Parameter. Ansonsten sind die Unterschiede minimal.



Jetzt schreiben wir Werte in die eindimensionale Matrix, schön übersichtlich. Wir holen uns zuerst die Dimensionen und die Daten und bestimmen die breite der Matrix – die steht als einziger Wert in den Dimensionen. Wenn es sich um eine eindimensionale Matrix handelt und der Index im richtigen Bereich liegt, schreiben wir den neuen Wert an die richtige Stelle. Im zweidimensionalen Fall handeln wir entsprechend.



Die Zugriffsoperationen (get…) arbeiten entsprechend.



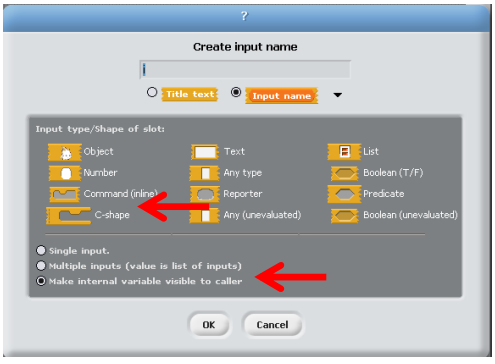
Die Benutzung dieser neuen Strukturen entspricht den Vorgaben.

Schöner wäre es, wenn wir über eine Zählschleife verfügten, mit der sich Matrizen einfach beschreiben und auslesen lassen. Das ist in BYOB leicht möglich, weil sich Skripte als Daten verarbeiten lassen.[21] Die Struktur soll sein:

for <variable> from <anfang> to <ende> do
<was hier steht>

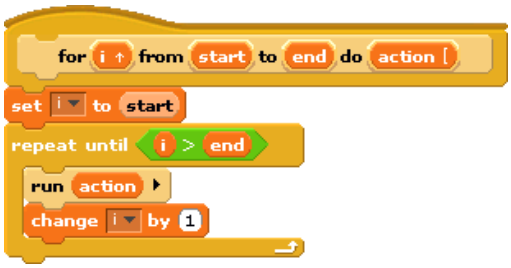
Wir schreiben also wie üblich einen Kopf eines neuen Blocks, den wir der Control-Rubrik zuordnen. Zwischen „for“ und „from“ fügen wir einen Parameter i ein, den wir als „Make internal variable visible to caller“ typisieren. Damit können wir beim Aufruf der Zählschleife den Namen der Zählvariablen „von außen“ ändern.

Vor und nach „to“ kommen zwei Parameter start und end, die als Zahlen typisiert werden, und am Ende folgt ein Parameter action. Diesen typisieren wir als C-shape um festzulegen, dass hier ein Skript eingefügt werden kann.

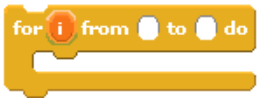


[21] Ab jetzt weitgehend nach dem BYOB-Manual von Brian Harvey und Jens Mönig.

Der Aufbau dieses Blocks ist einfach. Das Skript action wird so oft wie nötig mit run ausgeführt.



In der Control-Rubrik finden wir die Zählschleife als neuen Block. Wir können sie wie üblich anwenden, schachteln usw.

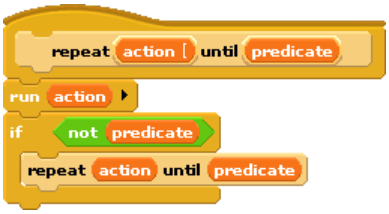


7.5 Ausdrücke mit verzögerter Evaluation

Als Beispiel für eine etwas fortgeschrittene Programmierung in BYOB wollen wir eine fußgesteuerte Schleife einführen:

repeat <action>
until <predicate>

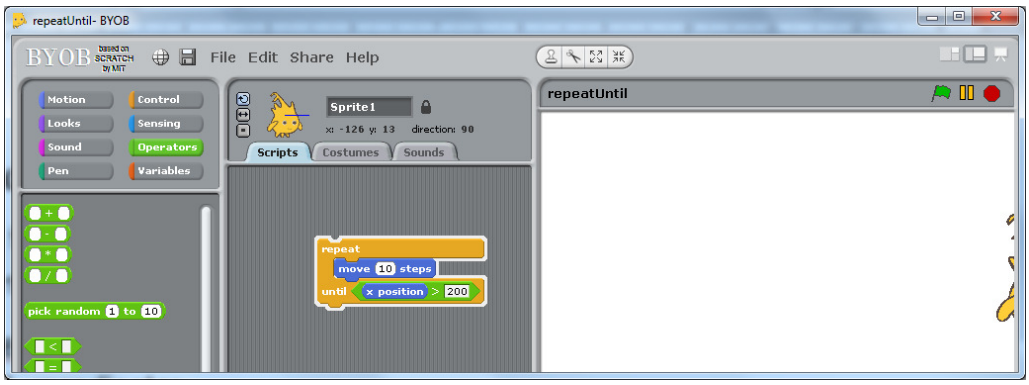
Wir definieren dafür einen entsprechenden Block und fügen zwei Parameter action (typisiert als C-shape) und predicate (typisiert als boolean) ein. Den Rumpf des Blocks formulieren wir schön rekursiv.



der neue Block

Mit diesem neuen Block formulieren wir ein einfaches Skript:

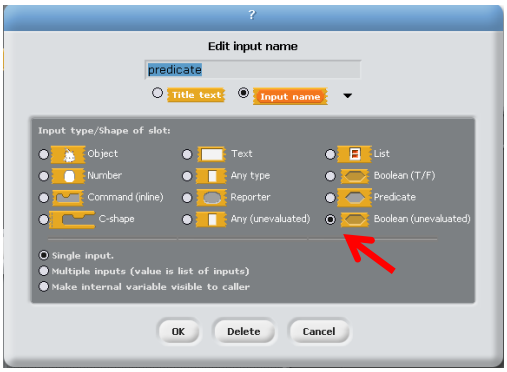
Das Ergebnis ist leider nicht wie erwartet – Alonzo rennt „bis vor die Wand“ und hat dann eine x-Koordinate von 274, also deutlich mehr als 200.



Woran liegt das?

Nach einem sorgfältigen Programmtest z. B. mit Kontrollausgaben und/oder einer geeigneten Trace-Tabelle stellen wir fest, dass das Prädikat immer den gleichen Wert hat (hier: false), auch wenn Alonzo längst die 200er-Grenze überschritten hat. Der Grund ist einfach: die Werte von Parametern werden berechnet (evaluiert), bevor sie dem aufgerufenen Block übergeben werden. In unserer neuen Schleife muss aber immer wieder nachgesehen werden, ob das Prädikat schon erfüllt ist.

Für diesen Zweck gibt es im Typisierungsfenster einige zusätzliche Optionen zu nicht evaluierten Ausdrücken. Ein Parameter dieser Art wird mit einem Stern hinter dem Namen gekennzeichnet. Sein Wert wird vor dem Methodenaufruf nicht evaluiert, sondern der dafür erforderliche Code wird übergeben, sodass er im Block bei Bedarf ausgeführt werden kann. Das muss er dann aber auch!



Verwendung von „special forms“

Da wir einen Wahrheitswert benötigen, wählen wir den Typ Boolean (unevaluated) für unseren pedicate-Parameter aus. Damit funktioniert unser neuer Block aber immer noch nicht richtig, denn die Auswertung des Prädikats muss ja irgendwann geschehen! Das ist in unserem Fall an zwei Stellen der Fall: um zu entscheiden, ob die Rekursion fortgesetzt wird und bei der Übergabe der Parameter. In beiden Fällen rufen wir das als Skript übergebene Prädicat mit call auf – das im zweiten Fall wiederum als Skript interpretiert wird.

bei Bedarf müssen diese evaluiert werden

Nach diesen Änderungen haben wir eine schöne neue fußgesteuerte Schleife im Werkzeugkoffer.

7.6 Aufgaben

1. Informieren Sie sich im Netz über die verschiedenen Sortierverfahren. Implementieren Sie einige davon wie Shakersort, Gnomsort, Insertionsort, …

2. Implementieren Sie dreidimensionale Matrizen mit den entsprechenden Zugriffsmethoden.

3. Implementieren Sie Matrizen anders, indem Sie die verwendeten Listen anders strukturieren.

4. Erzeugen Sie neue Kontrollstrukturen:
   a: für eine rückwärts zählende Schleife:
      for <..> from <..> downto <..> do <action>
   b: für eine kopfgesteuerte Schleife
      while <prädikat> do <action>
   c: für eine weitere fußgesteuerte Schleife
      do <action> while <prädikat>

5. Implementieren Sie eine Mehrfachauswahl
      switch <wert> <Möglichkeiten>
   a: Diskutieren Sie mindestens zwei unterschiedliche Möglichkeiten, diese Kontrollstruktur zu implementieren.
   b: Realisieren und testen Sie mindestens eine davon.

6. a: Informieren Sie sich über die Datenstruktur dictionary.
   b: Implementieren Sie die Struktur mit geeigneten Zugriffsoperationen.

7. a: Implementieren Sie die Datenstruktur Stapel.
   b: Implementieren Sie die Datenstruktur Schlange.

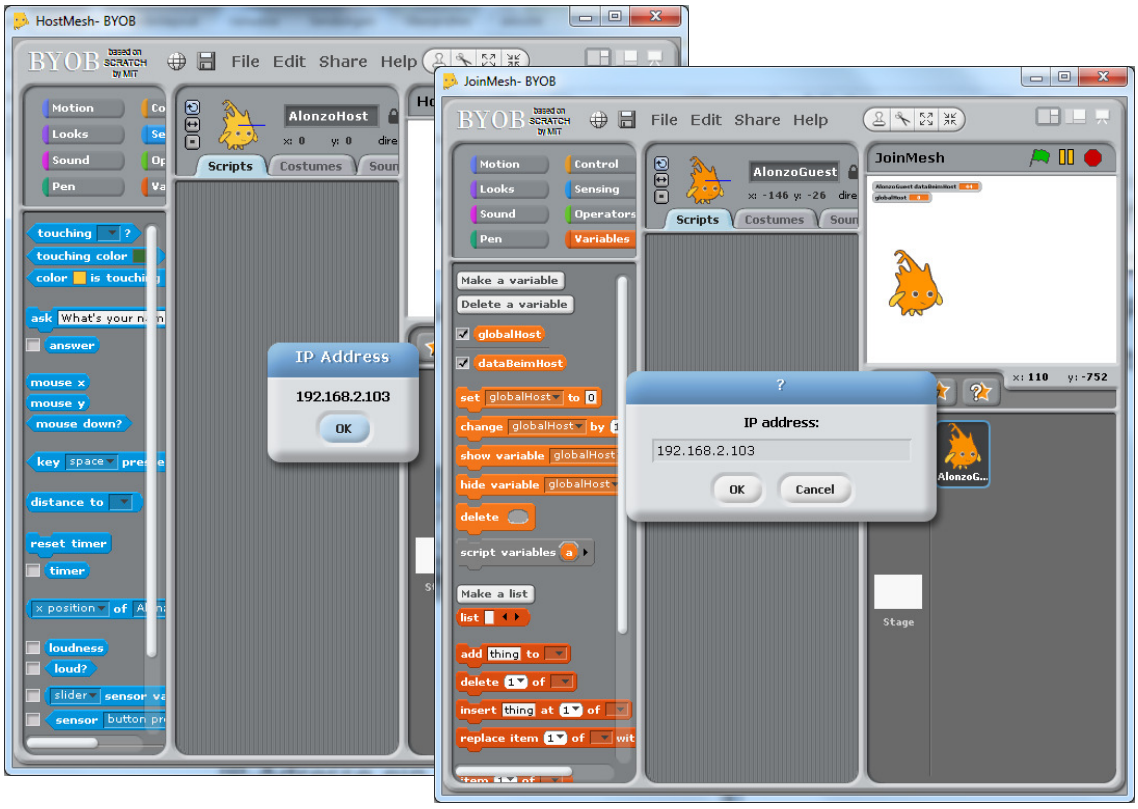
8. Implementieren Sie einen einfachen Binärbaum mit den Operationen
   a: new Baum
   b: rein <element> in <baum>
   c: zähle die Elemente in <baum>
   d: ist <element> vorhanden in <baum>?
   e: raus <element> aus <baum>
   f: ermittle die maximale Tiefe von <baum>
   g: balanciere <baum> aus

8 Im Netz arbeiten

BYOB enthält eine Mesh-Funktionalität, die rudimentären Zugriff auf andere BYOB-Instanzen, auch auf anderen Computern, gestattet. Wir nutzen diese für einige kleine Anwendungen aus.

8.1 Funktionen von Mesh

Wir starten zwei BYOB-Instanzen, egal ob auf demselben Rechner oder auf verschiedenen im gleichen Netz, und wählen den Menüpunkt Share im Hauptmenü. Dort finden wir die Menüpunkte Host Mesh und Join Mesh. Das Programm in der ersten BYOB-Instanz speichern wir unter dem Namen HostMesh, das andere als JoinMesh. In zweiten färben wir Alonzo ein und nennen ihn AlonzoGuest, im ersten AlonzoHost. Wir klicken im ersten Host Mesh an und erhalten ein Fenster mit der IP-Adresse des Rechners. Im zweiten klicken wir Join Mesh an und geben die angezeigte IP-Adresse ein.

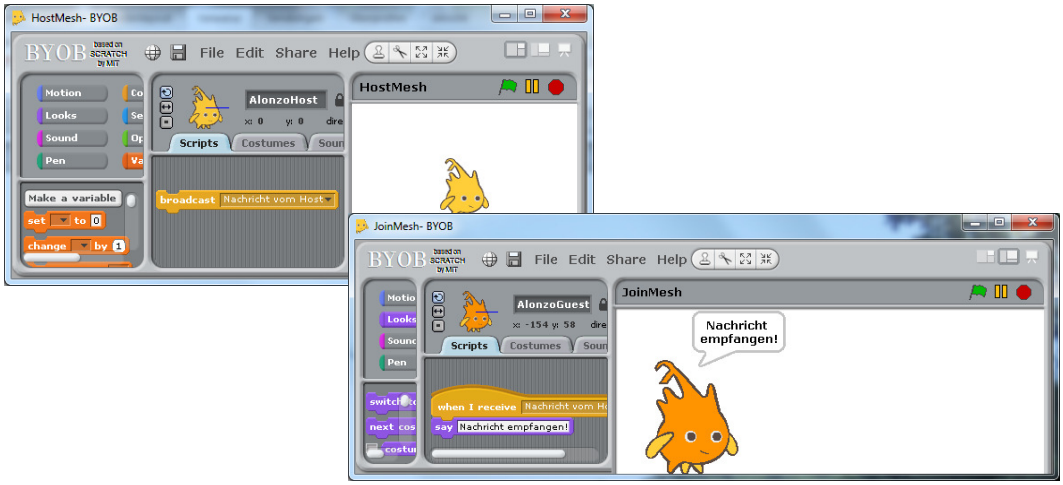


zwei Instanzen von BYOB, über Mesh verbunden

Die entstandene Verbindung ermöglicht nun eine sehr stark vereinfachte Kommunikation zwischen den Instanzen:

1. Botschaften werden in beiden BYOB-Instanzen empfangen.
2. Variable der anderen Instanz können über die Kachel <slider> sensor value erreicht werden.
3. Sprites können „geshared“ werden und sind dann in der anderen Instanz sichtbar.

zu 1) Wir senden in der einen BYOB-Instanz eine Botschaft. Auf diese kann in beiden Instanzen reagiert werde. Allerdings besteht ein kleiner Unterschied, ob ein Sprite „geshared“ wurde oder nicht: im ersten Fall werden die möglichen Botschaften angezeigt, im anderen muss die Nachricht, auf die reagiert wird, genauso wie sie gesendet wird eingegeben werden.



zu 2) Erzeugt man in einer der BYOB-Instanzen eine globale Variable, dann kann auf diese über 

in der anderen Instanz zugegriffen werden. Ändert die Variable ihren Wert, dann wird der geändert auch in der lesenden Instanz richtig ermittelt. Wird die Variable gelöscht, dann bleibt sie allerdings in der anderen im Zugriff mit dem letzten Wert erhalten.

Achtung! BYOB-Fehler

Gelöschte Variable bleiben in der anderen Instanz sichtbar.

zu 3) „Shared“ man ein Sprite, dann wird es in der anderen Instanz sichtbar, mit allen seinen Skripten. Diese können allerdings nicht mit Wirkung auf das Original ausgeführt werden, aber schon mit Wirkung auf die Kopie. Die nimmt alle seine lokalen Variablen mit, ist dann aber vom Original getrennt. Mit Ausnahme des direkten Zugriffs auf alle Botschaften des Originals bringt das „Sharen“ von Sprites also keinen rechten Vorteil, denn die Kopie könnte man auch über den Import und Export von Sprites realisieren.

Wir werden uns deshalb auf den Austausch von Botschaften und den Zugriff auf globale Variable beschränken.

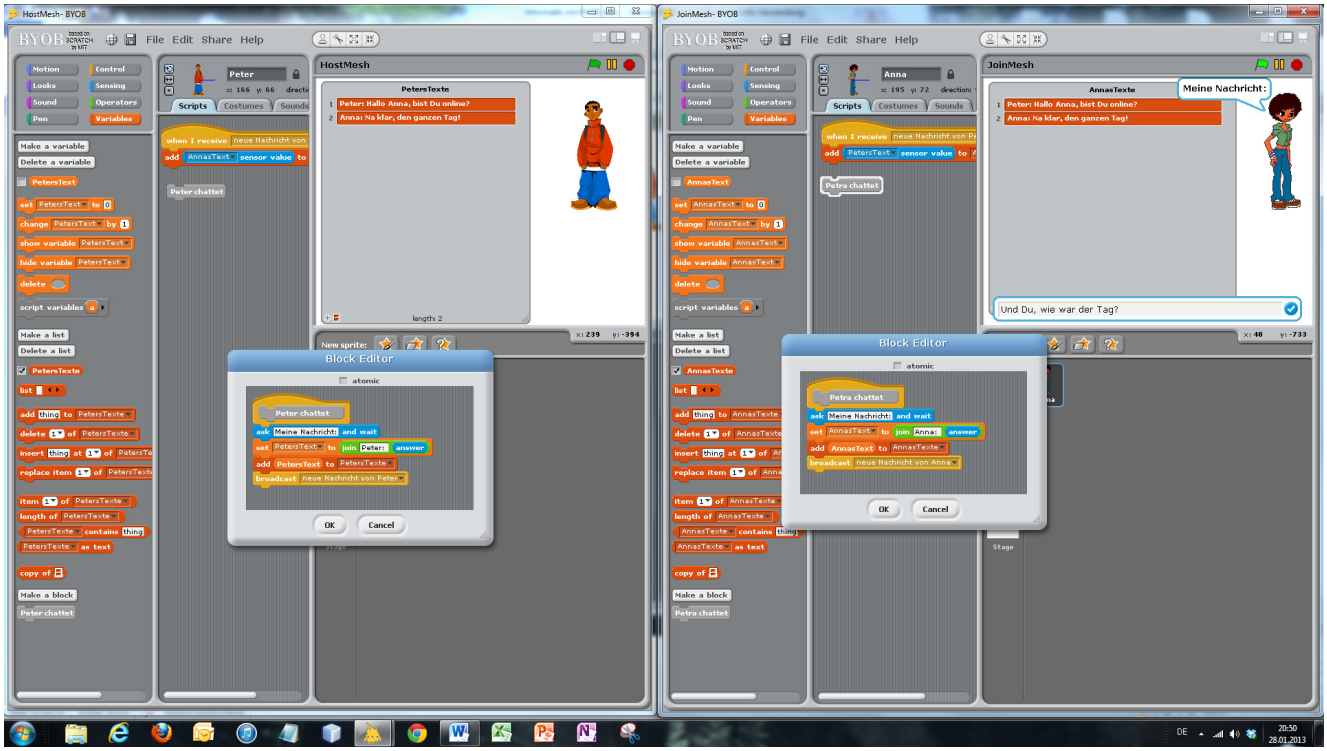
8.2 Chatten

Wir starten wie beschrieben zwei BYOB-Instanzen, löschen jeweils Alonzo und laden ein Sprite – einmal einen Jungen, einmal ein Mädchen. Wir nennen sie Peter und Anna. Die beiden sollen miteinander chatten. Dazu werden die BYOB-Instanzen über Mesh verbunden.

Wir wählen die folgenden Regeln für die Kommunikation:

- Die jeweils neue Textzeile wird in zwei globalen Variablen namens PetersText bzw. AnnasText abgelegt. Diese Variablen können von der jeweils anderen BYOB-Instanz gelesen werden.
- Die Darstellung des Chatverlaufs erfolgt in zwei Listen PetersTexte und Annas-Texte, die in den jeweiligen Bildbereichen groß dargestellt werden. Da es sich hier nicht um Variable handelt, sind diese Listen für die jeweils anderen Partner nicht sichtbar und werden selbst ergänzt.
- Wird eine neue Textzeile eingegeben, dann wird nach der Eingabe dieser der Name des Senders vorangestellt. Danach wird diese Kombination in die eigene Liste sowie globale Variable kopiert und eine Nachricht ausgesandt, dass ein neuer Text bereitstellt.
- Auf diese Nachricht reagiert der Partner, indem er sie an seine Liste anhängt.

Die (sehr kurzen) Skripte und eine Beispielkommunikation sind unten dargestellt.



8.3 Aufgaben

1. Verbessern Sie die Funktionalität des Chatprogramms, indem Sie Buttons zur Bedienung, eine Möglichkeit zum Löschen aller Texte / der letzten Zeile / ausgewählter Zeilen / … einführen.

2. Erweitern Sie das Chatprogramm um die Möglichkeit, mehrere Partner in den Chat aufzunehmen.

3. Chats sollen natürlich privat bleiben. Verschlüsseln und entschlüsseln Sie die Daten mit einem Password, sodass nur Teilnehmer/innen mit Kenntnis des Passwords die Klartexte sehen können.

4. Eine BYOB-Instanz soll einen Bankautomaten darstellen, die andere die Bank.
   a: Verteilen Sie die Funktionalität geeignet.
   b: Diskutieren Sie Möglichkeiten zu einer sicheren Kommunikation zwischen den beiden.
   c: Realisieren Sie das Projekt.

5. Eine BYOB-Instanz stellt einen Online-Shop dar, mehrere andere die Kunden. Behandeln Sie das Problem so wie in Aufgabe 4.

6. Zwei Spieler sollen ein Spiel[22] miteinander im Netz spielen können. Behandeln Sie das Problem wie in Aufgabe 4.

7. Eine BYOB-Instanz stellt einen Mailserver dar, mehrere andere die Clients. Behandeln Sie das Problem wie in Aufgabe 4 unter besonderer Berücksichtigung der Sicherheit.

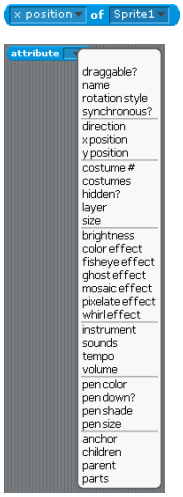
8. Entwickeln Sie eine Möglichkeit zum verteilten Rechnen im Netz. Ein Server verwaltet mehrere Clients, die arbeitsteilig ein Problem lösen können.
   a: Informieren Sie sich im Netz über mögliche Einsatzgebiete verteilten Rechnens und entsprechende Projekte.
   b: Diskutieren Sie die Eignung unterschiedlicher Problemklassen für diese Aufteilung. Gehen Sie dabei auch auf die funktionale Programmierung ein.
   c: Realisieren Sie ein Projekt.

9. Mehrere BYOB-Instanzen stellen ein sternförmiges Netzwerk dar mit einem einzigen Server. Die Clients können Daten entweder an Sprites innerhalb der eigenen Instanz adressieren oder an Sprites in anderen Instanzen senden.
   a: Entwickeln Sie geeignete Adressierungsarten, anhand derer die Clients entscheiden können, ob Datenpakete an eigene Sprites oder andere auszuliefern sind.
   b: Entwickeln Sie ein entsprechendes Protokoll und realisieren Sie es.

[22] z. B. „Schiffeversenken“

9 Objektorientierte Programmierung

Zugriff auf Attribute



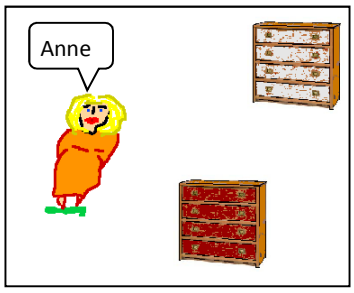
BYOB arbeitet natürlich die ganze Zeit mit Objekten[23], die hier *Sprites* genannt werden. Sie verfügen über eigene Attribute (z. B. Position, Richtung, Kostüm, …), auf die mithilfe unterschiedlicher Blöcke zugegriffen werden kann. Der attribute <auswahl>-Block liefert die ganze Palette. Weiterhin können Sprites über eigene Methoden verfügen, wenn man sie für sie schreibt.

Ein wesentlicher Aspekt der OOP ist die Vererbung. Diese kann in BYOB sehr unterschiedlich gehandhabt werden. Für den Unterricht am interessantesten ist Liebermanns[24] Delegation-Modell, das mit Prototypen (also konkreten Objekten, nicht abstrakteren Klassen) arbeitet und diese bei Bedarf klont. Das Vorgehen ergänzt das übliche, top-down-orientierte Vorgehen um eine bottom-up-Komponente. Wir werden hier beides vorstellen.

9.1 Die Kommunikation zwischen Objekten

Als Beispiel erzeugen wir (als Sprite) einen einfachen Datenspeicher mit einer lokalen Liste Inhalte, den wir durch eine Kommode repräsentieren. Diesen statten wir mit lokalen Zugriffsmethoden auf die Daten aus, indem wir die Methoden rein <daten> und raus implementieren. Wir erhalten eine simple Queue. So können wir zwar beliebige Inhalte in die Liste schreiben und daraus entfernen, aber ausreichend ist das nicht, weil ein Datenspeicher sich ja nicht selbst befüllt. Wir benötigen also einen Zugriff auf die Methoden des Objekts von außen.

Um das zu demonstrieren, erzeugen wir zwei Kopien[25] unseres Speicher-Sprites, die wir Akten und Andenken nennen, sowie eine IT-Beauftragte Anne und erhalten die nebenstehende Konfiguration.



Wie kann Anne auf ihre Datenspeicher zugreifen?

In BYOB sind die Scratch-Blöcke <attribute> of <sprite> und set <variable> to <value> stark modifiziert, und zwei neue, object und attribute, sind hinzu gekommen. Mit diesen können wir auf die Attribute und Methoden eines anderen Objekts zugreifen, wenn wir uns klarmachen, dass BYOB nur mit „erstklassigen“ Komponenten umgeht. So können auch Skripte entweder als Programme oder als Daten angesehen und in unterschiedlichen Kontexten evaluiert werden.

---

[23] weitgehend nach Modrow, Objektorientierte Programmierung mit BYOB, LOG IN 171 (2012)

[24] Lieberman, Henry: Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems, ACM SIGPLAN Notices, Volume 21 Issue 11, Nov. 1986

[25] Kopien enthalten schon anfangs ihre eigenen Attribute, hier die Liste. Erzeugten wir Klone, dann würden alle mit der gleichen Liste arbeiten, die wir erst überschreiben müssten, um unabhängige Speicher zu erhalten.

Wählen wir im Skriptbereich von Anne im Block <attribut> of <sprite> einen der Aktenschränke, dann erscheinen dort zusätzlich zu den Standardattributen die lokalen Methoden des anderen Sprites: Anne kann also „sehen“, was ein Aktenschrank kann. Wählen wir die Methode raus und klicken dann auf den Block, dann werden wir allerdings enttäuscht: das Ergebnis der Operation ist nicht etwa ein gespeichertes Element des Aktenschranks, sondern die Methode raus selbst.[26] Das ist auch sinnvoll, denn würde die Methode im Kontext von Anne evaluiert, erhielten wir einen Fehler: Anne verfügt über kein solches Skript. Wir müssen also das übergebene Skript mithilfe des call-Blocks im richtigen Kontext ausführen, und das ist der des Aktenschranks. Ist der noch leer, dann erhalten wir die entsprechende Antwort.

Zugriff auf lokale Methoden

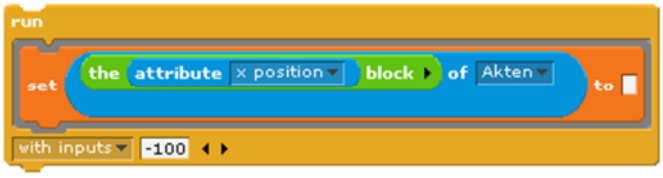
Auf eine ähnliche Weise schreiben wir Daten in den Aktenschrank: wir rufen das richtige Skript im richtigen Kontext auf. Da es sich bei rein um einen command-Block handelt, führen wir diesen mit run aus und übergeben als Parameter (input) die zu speichernden Daten.

Die Sache wird klarer, wenn wir sie in Einzelschritte zerlegen:

1. Anne sieht bei einem Objekt nach, über welche lokalen Methoden dieses verfügt.
2. Anne wählt eine dieser Methoden aus und weist sie einer Variablen namens Skript zu. Das geht, weil Skripte erstklassig sind und genau dieses zulassen.

3. Handelt es sich um eine Funktion (reporter), dann sendet Anne dem Objekt mithilfe des call-Blocks die Botschaft, die im Skript gespeicherte Operation auszuführen. Bei Bedarf fügt sie die notwendigen Parameter hinzu (call with inputs). Das Ergebnis des Funktionsaufrufs kann sie dann weiter verarbeiten. Handelt es sich um eine Anweisung (command), dann verfährt sie entsprechend mithilfe des run-Blocks.

Wollen wir darüber hinaus auch Attribute eines Sprites von außen verändern, dann kann der in BYOB erweiterte set-Block benutzt werden – aber auch wieder im richtigen Kontext: wir rufen ihn mit run dort auf.



Ändern von Attributwerten. Der graue Rand signalisiert auch hier, dass der innere Block als Skript interpretiert wird.

[26] Man erkennt das auch am grauen Rand um den Skriptnamen. Wird das Ergebnis eines Skripts benötigt, z. B. beim Einfügen in einen Block, dann muss ein weißer Rand erscheinen. Wird das Skript selbst geliefert, dann erscheint der graue – manch mal aus Versehen! Deutlicher wird das Ganze, wenn man die the skript - und the block-Blöcke benutzt.

Anne als gut ausgebildete IT-Beauftragte kann solche Befehle natürlich absetzen, ein normaler Benutzer aber wohl nicht. Anne stellt deshalb neue Blöcke zur Verfügung, die als Parameter zusätzlich den zu benutzenden Aktenschrank erhalten. Damit wird die Benutzung sehr vereinfacht. Auf ähnliche Weise organisiert sie den Zugriff auf gespeicherte Daten.

9.2 Aufgaben

1. Implementieren Sie bei den Aktenschränken selbst oder bei der IT-Beauftragten eine Zugriffskontrolle
   a: durch eine Passwordabfrage.
   b: mit Benutzerlisten und zugeordneten Passwords.

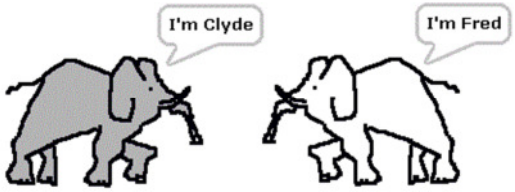
2. Verarbeiten Sie die Daten selbst, indem Sie
   a: Plausibilitätsprüfungen einführen.
   b: Verschlüsselung einführen.
   c: Organisationsformen in Listen, Reihungen, Stapeln, Schlangen, Bäumen usw. einführen.

3. Verteilen Sie die Funktionalität mithilfe der Mesh-Möglichkeiten von BYOB auf unterschiedliche Instanzen von BYOB, die auf unterschiedlichen Rechnern laufen.

9.3 Vererbung durch Delegation

Bisher haben wir als Prototypen eines Datenspeichers unsere Kommode, von der wir Kopien ziehen, an die wir – wie in anderen OOP-Sprachen auch – Botschaften in Form von Methodenaufrufen senden. Zur OOP gehört aber zentral das Konzept der Vererbung.

Im Originalartikel von Lieberman werden Objekte als Verkörperung der Konzepte ihrer Klasse verstanden. So steht dort der Elefant Clyde für alles, was der Betrachter unter einem Elefanten versteht. Stellt sich dieser einen Elefanten vor, dann erscheint vor seinem geistigen Auge nicht etwa die abstrakte Klasse der Elefanten, sondern eben Clyde. Spricht er über einen anderen Elefanten, hier: Fred, dann beschreibt er diesen etwa so: *„Fred ist genauso wie Clyde, bloß weiß.“*



Was bedeutet dieser Ansatz für den Lernprozess? Kennt der Lernende nur ein Exemplar einer Klasse (hier: Clyde), dann beschreibt der Prototyp seine Kenntnisse vollständig, eine Abstraktion ist für ihn sinnlos. Lernt er danach andere Exemplare kennen und beschreibt diese durch Modifikationen am Original, ersetzt also einige Methoden durch andere, verändert Attribute und ergänzt neue, dann entsteht langsam das Bild der Klasse selbst als Schnittmenge der gemeinsamen Eigenschaften. Erst jetzt ist der Abstraktionsvorgang für ihn nachvollziehbar und nach einigen Versuchen auch selbst gangbar. Delegation ist damit ein Verfahren, das den Lernprozess selbst abbildet, indem statt Klassen Prototypen erstellt werden.

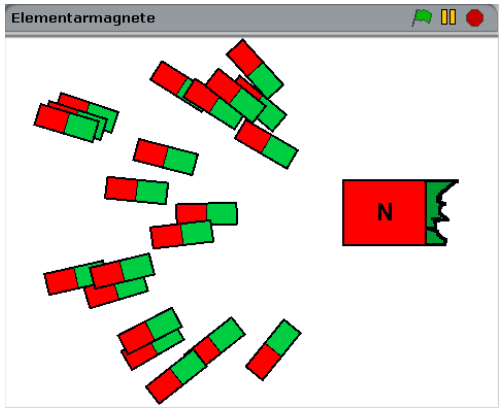
Klonen von Sprites

In BYOB werden Sprites als Prototypen erzeugt und mit den gewünschten Attributen und Methoden ausgestattet. Ist deren Verhalten genügend erprobt worden, dann können Klone entweder statisch im Sprite-Fenster oder dynamisch mithilfe des clone-Blocks erzeugt werden. Für jedes Sprite kann angezeigt werden, von welchem Sprite es abgeleitet wurde (parent) und über welche Kinder es verfügt (children…). Die Parent-Eigenschaft kann auch nachträglich gesetzt und/oder verändert werden, sodass das System der Abhängigkeiten dynamisch ist.

Ein Klon erbt anfangs alle Attribute und Methoden des Mutterobjekts. Angezeigt wird dieses durch eine „blassere“ Darstellung in den Paletten. Überschreibt ein Sprite geerbte Attribute oder Methoden, dann ersetzen diese wie üblich diejenigen des Prototypen. Löscht man die Überschreibungen wieder (ggf. auch dynamisch mithilfe des delete-Blocks), dann erscheinen erneut die geerbten.

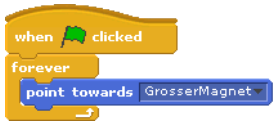
9.4 Magnete

Als ein sehr einfaches Beispiel für den Umgang mit Objekten wählen wir ein Magnetfeld, dessen Orientierung in der Nähe eines „Nordpols“ durch „Elementarmagnete“ angezeigt wird. Die kleinen Dinger sollen einfach auf den Nordpol zeigen.



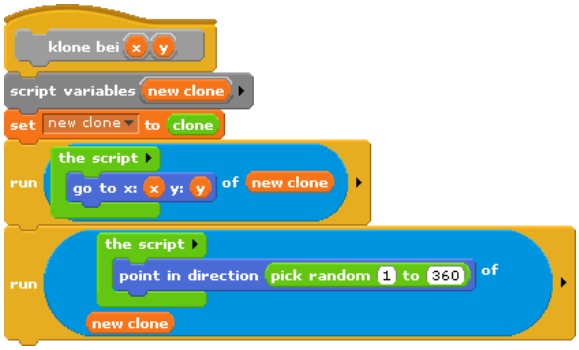
Wir zeichnen also den großen Magneten ohne jede Funktionalität (man kann ihn nur durch die Gegend schieben) und einen einzelnen kleinen. Diesen statten wir mit den erforderlichen Eigenschaften aus und klonen ihn so oft wie nötig.

Das Zeigen auf den Großen ist einfach. Etwas komplizierter ist das Klonen, weil wir die Klone natürlich im Bildbereich verteilen wollen, etwa so:



Wenn die kleinen Magnete im linken Bildbereich verteilt werden, dann könnte das so gehen – allerdings nur, wenn eine klone bei-Methode zur Verfügung steht. Die müssen wir jetzt – unter Ausnutzung der neuen Kenntnisse über Methodenaufrufe anderer Objekte – schreiben.

Wir erzeugen einen Klon und weisen den einer lokalen Variablen zu. Den Klon schicken wir dann an die Position, die durch die Parameterwerte angegeben wird, und drehen ihn in eine beliebige Richtung. Fertig.



Wir können jetzt ganz einfach viele Klone erzeugen. Aber wie wird man die wieder los?

Jeder Prototyp, der parent, verfügt über eine Liste seiner Kinder (children). Diese holen wir uns als Kopie und löschen alle darin vorkommenden Objekte. Man beachte, dass erst das Objekt (erster Eintrag in der Liste) mit delete <object> gelöscht wird und danach der Eintrag in der Liste selbst mit delete <n> of <list>.

9.5 Ein Digitalsimulator aus Klonen

Als kleines Projekt wollen wir einen Digitalsimulator schreiben, der als Bauteile *Schalter*, *LEDs* und *NAND*-Gatter enthält, die durch *Kabelstücke* verbunden sind. Die benötigten Teile werden aus den entsprechenden Prototypen durch klonen erzeugt. Die Verbindungen werden hergestellt, indem sich Kabel mit Ein- und Ausgängen oder auch untereinander „berühren“.

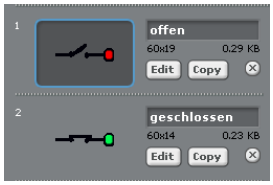
die Oberfläche

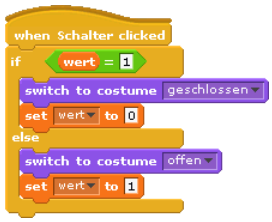
Wir setzen neben das Simulatorsymbol entsprechende Buttons und einen zusätzlichen mit der Aufschrift „X“, der alle erzeugten Klone löscht.

Bevor wir mit Klone erzeugen können, benötigen wir natürlich Prototypen.

Wir beginnen mit dem Schalter. Dieser hat zwei Kostüme, die den beiden Schalterzuständen entsprechen. Wie in der digitalen Elektronik üblich steht der Ausgang eines offenen Schalters auf logisch 1, der geschlossenen auf 0. Weiterhin verfügt er – wie fast alle folgenden Bauteile – über ein lokales Attribut wert.

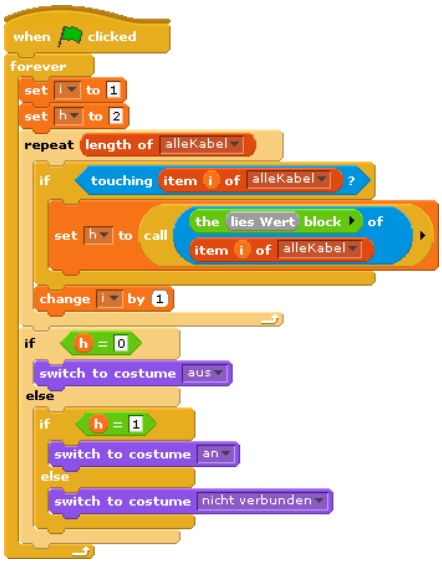
Schalter





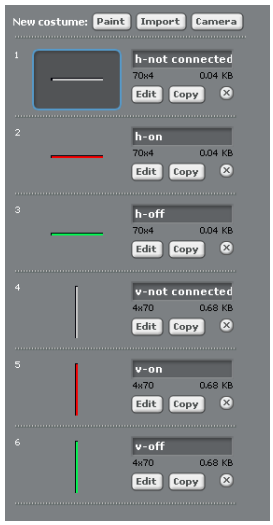
Wird beim Start der Simulation die grüne Flagge angeklickt, dann werden Schalter in den (offenen) Anfangszustand gebracht. Wird ein Schalter angeklickt, dann wechselt er den Zustand.

Entsprechend einfach ist das Verhalten der LEDs. Auch diese hat drei Kostüme. Bei Betrieb durchsucht sie die Liste der vorhandenen Kabel, alleKabel, und prüft, ob sie eines berührt. Ist das der Fall, übernimmt sie dessen Wert und zeigt ihn an.

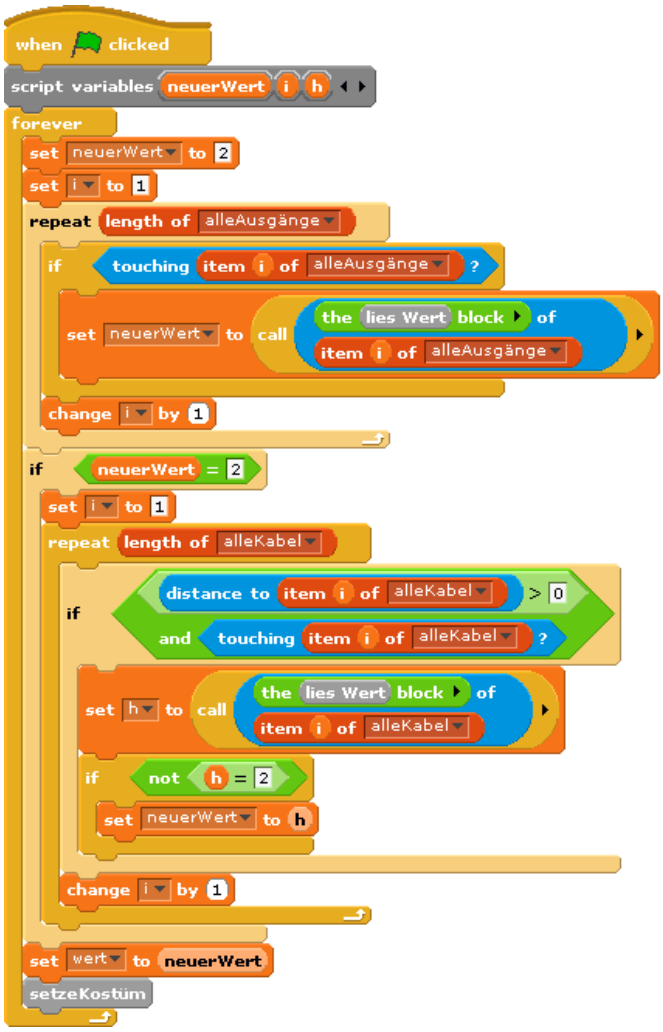


Etwas komplizierter ist der Umgang mit den Kabeln. Diese sollen beim Anklicken aus der vertikalen Lage in die horizontale kippen – und wieder zurück. Folglich benötigen sie sechs Kostüme.

Kabel

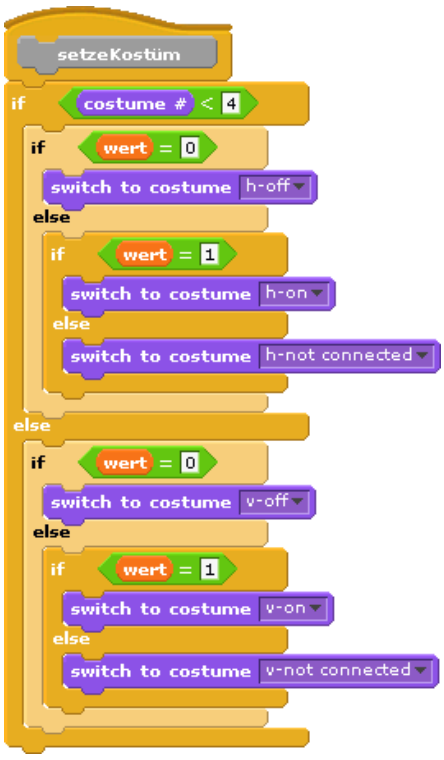


Vor allem aber sollen sie die Werte von Ausgängen oder anderen berührten Kabeln übernehmen.[27] Dazu durchsuchen sie zuerst die Liste aller Ausgänge. Berühren sie einen davon, dann merken sie sich deren Wert in der Variablen neuerWert. Berühren sie keinen Ausgang, dann probieren sie es mit den anderen Kabeln auf die gleiche Weise. Zuletzt übernehmen sie den bestimmten Wert und setzen ihr Kostüm auf den richtigen Wert.



Wie bestimmt man das Kostüm?

Nun, das ist einfach.



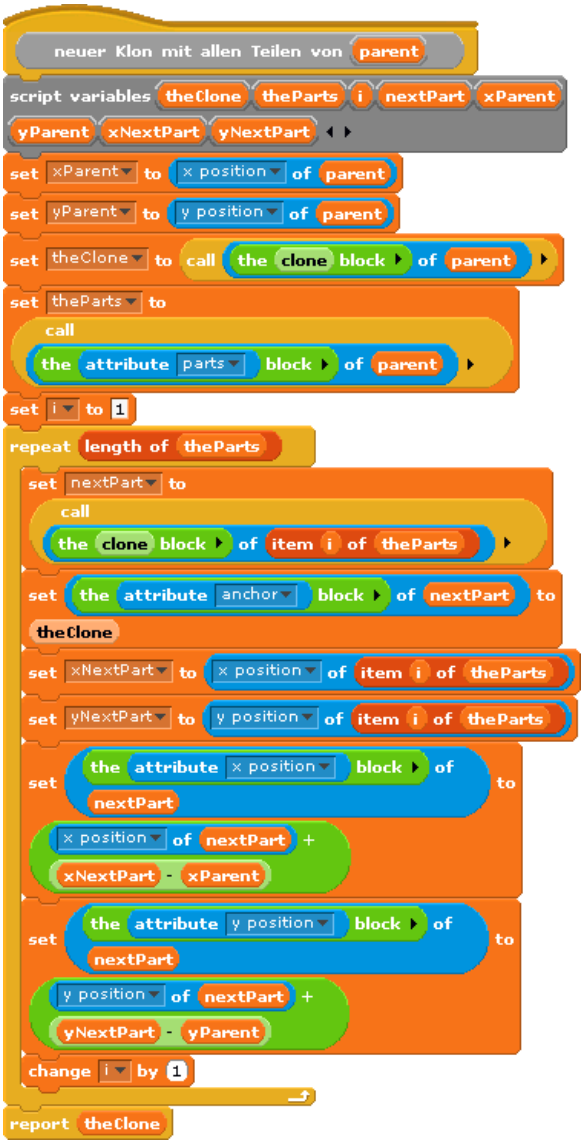
[27] Auf Feinheiten wie Kurzschlüsse usw. verzichten wir hier besser!

Jetzt kommen wir zu den Gattern. Ein Gatter besteht aus einem in der Simulation ziemlich funktionslosen Rumpf und einigen Buchsen, die je nach Verdrahtung reagieren. Die „schlauen“ Buchsen sollten also zusammen mit dem Rumpf eine Einheit bilden: eine Aggregation aus Objekten.

Sprites zusammenfügen

In BYOB erreichen wir das entweder interaktiv, indem wir ein Sprite aus dem Sprite-Fenster unten auf ein anderes Sprite auf der Arbeitsebene darüber ziehen. Dieses reagiert darauf sichtbar und fügt das untere Sprite in seine parts-Liste ein. Diese erreichen wir wie gewohnt über den attribute-Block. Hat das Gatter G1 also drei Buchsen als Teile, dann könnte das Ergebnis wie abgebildet aussehen.

Unser Anliegen besteht aber daraus, neue Gatter bei Bedarf dynamisch aus den Prototypen abzuleiten. Dazu müssen wir alle Teile eines Gatters durch Klonen erzeugen und die erforderlichen Beziehungen herstellen. Da wir unterschiedliche Gatter klonen wollen, übergeben wir den Prototypen als Muster und erstellen daraus ein neues Exemplar.



Der Prototyp parent wird als Parameter übergeben.

lokale Variable

Position des Parent bestimmen.

Klon des Parent erzeugen.

Liste der Teile des Parent bestimmen.

Mit allen Teilen tue …

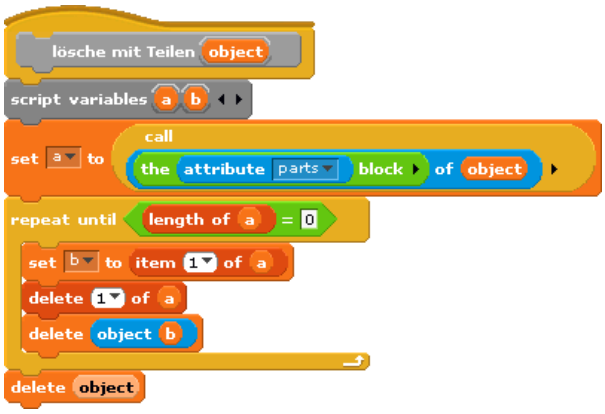
Klon des Teils erzeugen ....

… und mit dem Klon des Gatters verbinden.

Position des Teils relativ zum Gatterklon setzen.

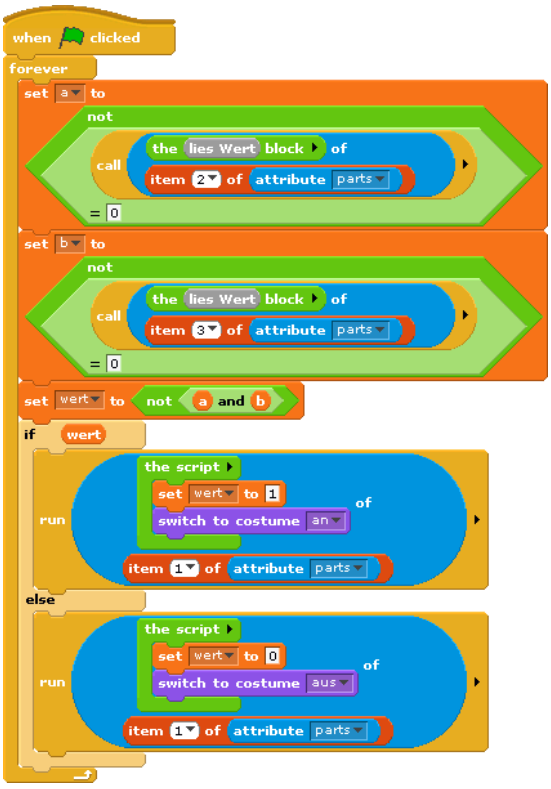
Ergebnis zurückgeben.

Wollen wir Klone löschen, dann müssen wir die verknüpften Teile mit abräumen. Es empfiehlt sich, dafür frühzeitig eine eigene Methode zu schreiben, weil sonst Massen von Klones „per Hand“ zu löschen sind. Das Beispiel demonstriert hoffentlich einsichtig den dynamischen Einsatz des delete-Blocks. Inhaltlich unterscheidet es sich nicht von der Arbeitsweise mit anderen Sprachen.

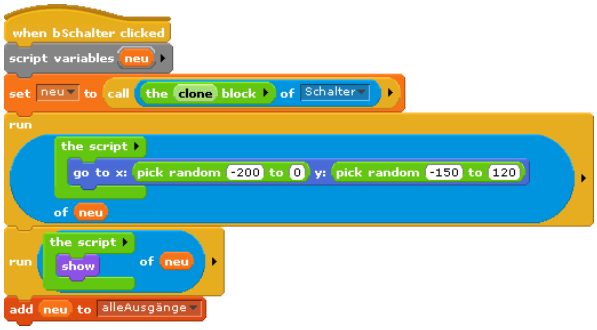


Funktionieren tut das Gatter noch nicht. Dafür lesen wir die Werte der Eingänge (Teile 2 und 3 in der parts-Liste) aus und bestimmen daraus den Wert des Ausgangs (Teil 1 in der parts-Liste).

Die globale lies Wert-Methode ermittelt den Wert der lokalen Variablen wert des aufgerufenen Objekts.

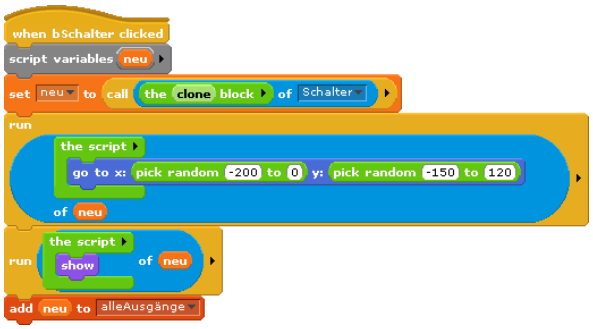


Jetzt kommen wir zu den Buttons. Diese müssen jeweils einen Klon des benötigten Objekts erzeugen. Wir schicken den dann zufällig an eine andere Stelle, um nicht alle Teile auf einem Haufen zu haben.

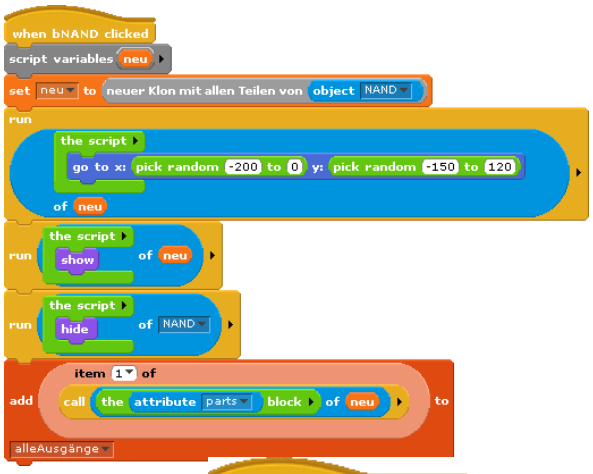


Für die Schalter sieht das wie nebenstehend aus, für die LEDs entsprechend.

Bei neuen Schaltern müssen wir zusätzlich dafür sorgen, dass sie in die Liste der Ausgänge eingetragen werden, …

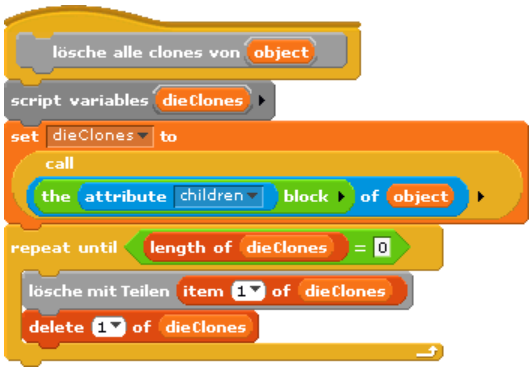


… und bei den Gattern ebenfalls.

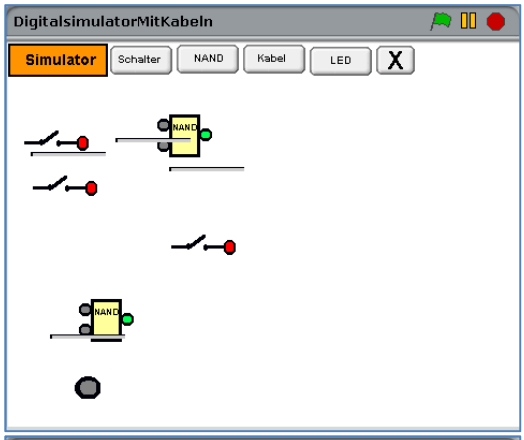


Und dann müssen natürlich mit dem X-Button alle vorhandenen Klone gelöscht werden!

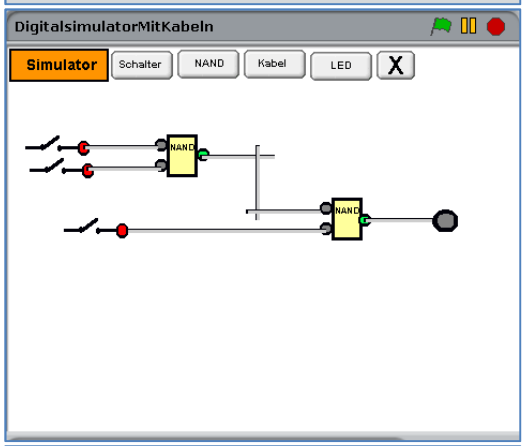
Ganz einfach, wenn man eine entsprechende Methode hat.



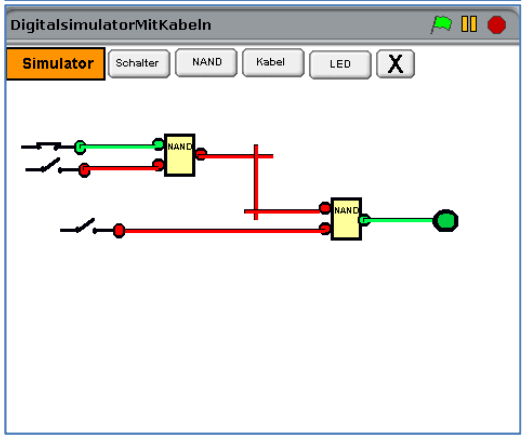
Das war es schon. Der Simulator ist zwar nicht gerade schnell, aber er arbeitet zufriedenstellend. Zur Simulation einer Schaltung erzeugen wir die benötigten Teile auf dem Bildschirm …



… schieben die Geräte in die richtige Position …



… und klickt die grüne Flagge an.

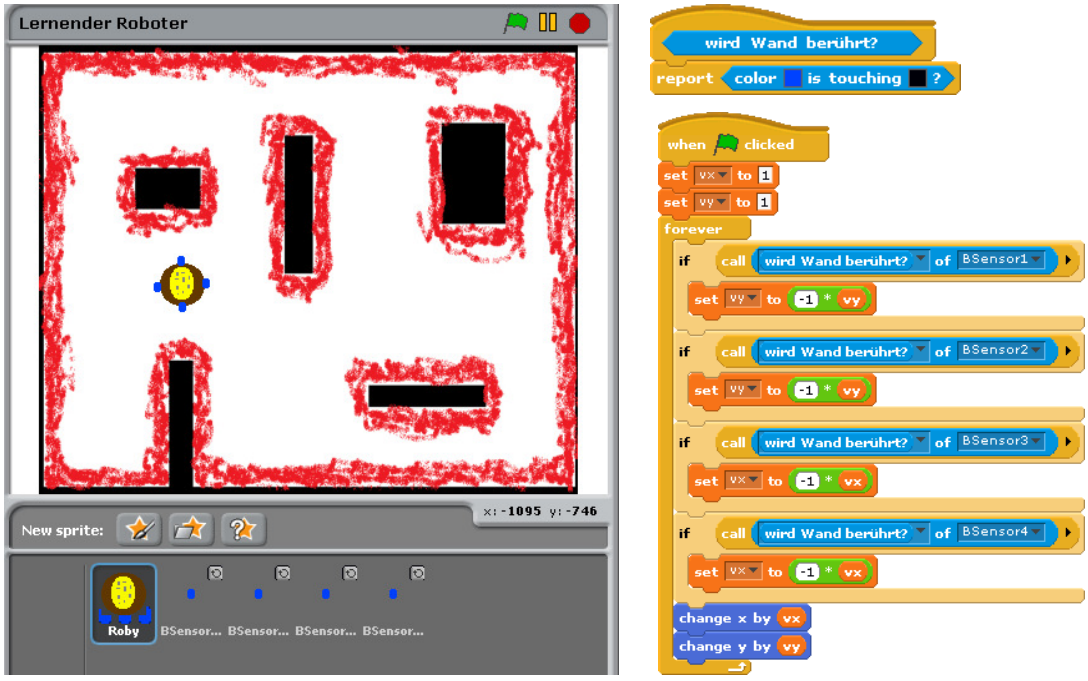


9.6 Ein lernender Roboter[28]

Als weiteres Beispiel für Vererbung durch Delegation wollen wir einen Roboter betrachten, der über vier Berührungssensoren verfügt. Kommt einer von diesen mit einem Hindernis in Berührung, dann ändert der Roboter seine Richtung, hat aber auch eine neue Beule.

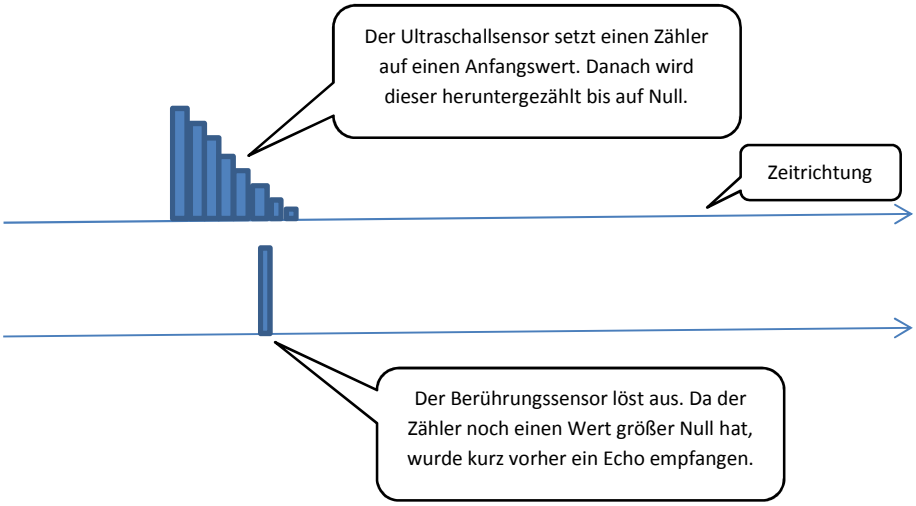
Wir zeichnen mit einem Zeichenprogramm ein Bild einer Welt, die von schwarzen Wänden begrenzt ist und in der einige schwarze Hindernisse stehen. Aus Gründen, die wir gleich kennenlernen werden, versprühen wir mit der Sprühdose einen diffusen roten Nebel um die Gegenstände herum und an den Wänden entlang. In diese Welt setzen wir Roby – als kleines kreisrundes Sprite. Weiterhin zeichnen wir ein kleines blaues Sprite, das wir mit einem Prädikat wird Wand berührt? ausstatten. Dieses Sprite klonen wir dreimal und heften die vier Sensoren dann an den Roboter. Es entsteht eine Aggregation. Den Roboter statten wir mit zwei lokalen Variablen vx und vy aus, die die Geschwindigkeitskomponenten in diesen Richtungen beschreiben. Meldet nun ein Berührungssensor eine Wand, dann wird die entsprechende Geschwindigkeitskomponente geändert.

Wir erhalten die folgende Konfiguration, in der sich Roby sicher zwischen den Hindernissen bewegt – wie gesagt, mit vielen Beulen:

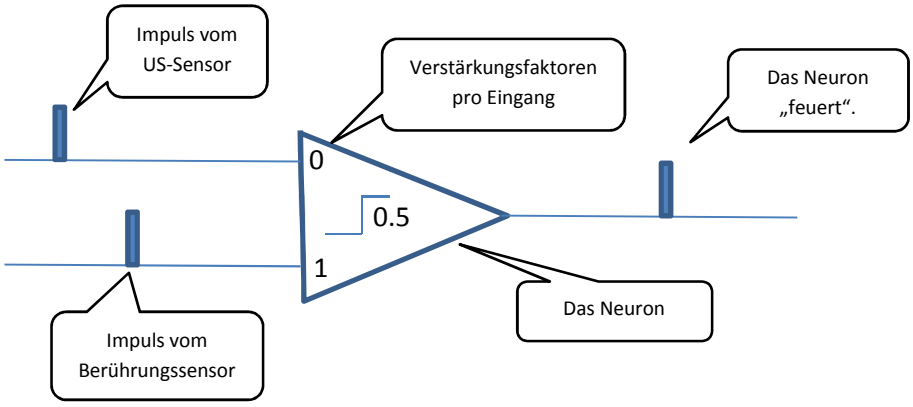


[28] Das Beispiel hat den Laufroboter von Prof. Florentin Wörgötter, Bernstein Zentrum für Computational Neuroscience Göttingen als Vorlage, beschrieben z. B. in http://www.chip.de/news/Schnellster-Roboter-lernt-bergauf-zu-gehen_27892038.html

Jetzt kommt die rote Sprühfarbe rund um die Hindernisse und Wände ins Spiel. Diese soll Bereiche kennzeichnen, in denen ein Ultraschallsensor Echos von den Gegenständen empfängt. Wir statten also den Roboter mit vier Ultraschallsensoren aus, die auf diese rote Farbe reagieren. Der Roboter soll lernen, dass ein Ultraschallecho oft einer Kollision vorausgeht, und dass es deshalb besser ist, schon bei diesem Echo umzukehren. Wir brauchen also einen Mechanismus, der feststellt, dass vor einer Kollision ein Echo kam. Eine Möglichkeit, dieses zu erreichen, ist ein Zähler im Ultraschallsensor, der auf einen Anfangswert (hier: 15) gesetzt wird, wenn er rote Farbe (also ein Echo) feststellt. Dieser Zähler wird kontinuierlich auf Null heruntergezählt – und ggf. schon vorher wieder heraufgesetzt. Hat dieser Zähler bei einer Kollision einen Wert größer als Null, dann ist das Echo kurz vorher empfangen worden.



Durch diese Konstellation wird ein Lernschritt in Gang gesetzt, der in einem Neuron stattfindet. Dieses verfügt über zwei Eingänge, die vom zugeordneten Berührungssensor bzw. Ultraschallsensor kommen und jeweils mit einem Gewicht behaftet sind, sowie einen Schwellwert. Die Leitung vom Berührungssensor hat das Gewicht 1. Kommt von dort ein Signal z. B. der Stärke 1, dann wird dieses mit dem Gewicht 1 multipliziert. Das Ergebnis ist größer als der Schwellwert (hier: 0.5) und das Neuron „feuert“. Das Gewicht des US-Sensors hat anfangs den Wert Null. Es wird immer dann erhöht, wenn der Berührungssensor bei einer Kollision feststellt, dass der Zähler des zugeordneten Ultraschallsensors einen Wert größer als Null hat. Finden genügend viele solcher kleinen Lernschritte statt, dann überschreitet das Produkt aus Gewicht und Signal auch beim US-Sensor den Schwellwert des Neurons und dieses feuert auch in diesem Fall.

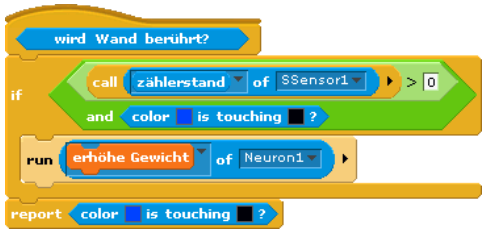


Diese Form Pawlowschen Lernens realisieren wir jetzt.

Der Ultraschallsensor arbeitet genau wie oben beschrieben. Der Zugriff auf das lokale Attribut zähler wird durch eine einfache Methode realisiert.

Die eigentlichen Änderungen finden also in den Berührungssensoren und den vier zugeordneten Neuronen statt. Da diese Klone des jeweils einzigen Prototypen sind, genügt es, nur in diesem die Ergänzungen vorzunehmen. Die Klone übernehmen diese, weil sie die Methoden des Prototypen erben.

Beim Berühren einer Wand muss noch festgestellt werden, ob der zugehörige Ultraschallsensor „kurz vorher“ ausgelöst hat. In den Klonen überschreiben wir danach die „blasse“ geerbte Methode, indem wir den zugeordneten Sensor verstellen. Damit verschwindet auch die Blässe. Vorher haben wir den Ultraschallsensor dreimal und das Neuron geklont und die vier neuen lila Ultraschallsensoren und die grünen Neuronen Roby angefügt. Der sieht jetzt so aus:

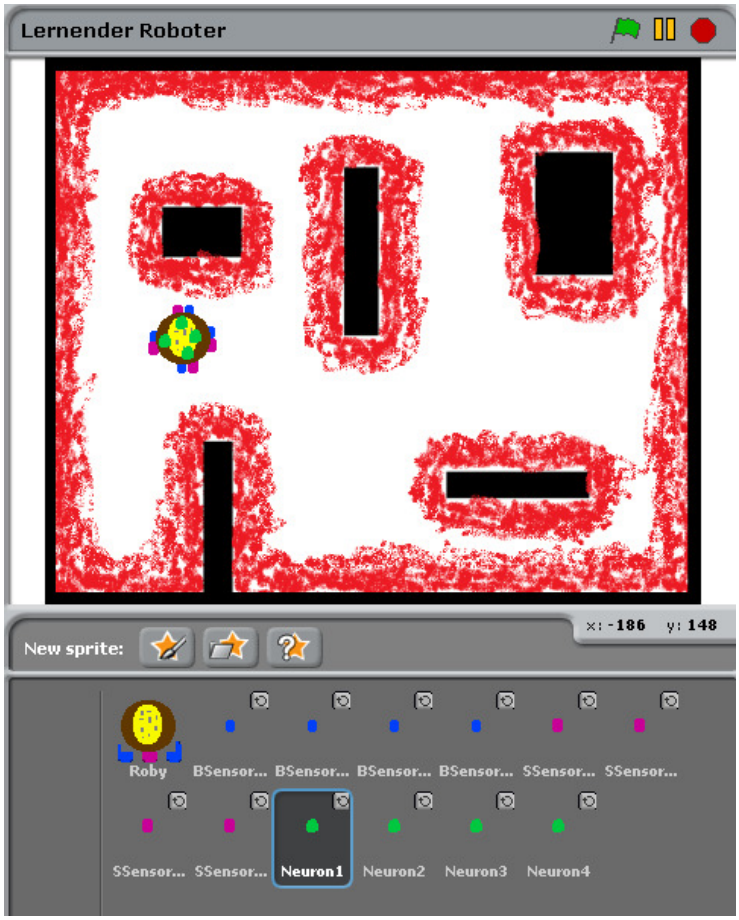


im Neuron

Das Neuron benötigen noch ein Prädikat feuert?, das wie beschrieben arbeitet.

Zuletzt ändern wir das Verhalten von Roby: der ändert seine Richtung, wenn das entsprechende Neuron feuert.

Roby sucht sich jetzt seinen Weg, anfangs zwischen den Hindernissen, dann entlang des „Echobereichs“.



9.7 Aufgaben

1. Verpassen Sie Roby eine Oberfläche, mit der sich die wesentlichen Faktoren ändern leicht lassen: seine Geschwindigkeit, die Gewichte, die Schwellwerte.

2. Führen Sie weitere Sensortypen, ggf. auch unter Einbeziehung des Picoboards, sowie weitere Ereignisse neben den Kollisionen ein.
   a: Lassen Sie Roby Korrelationen zwischen Sensorwerten und Ereignissen in unterschiedlichen „Welten“ finden. Roby passt sich so seiner Umgebung an.
   b: Diskutieren Sie Möglichkeiten, dass Roby sich an eine veränderliche Umwelt anpasst.

3. Diskutieren Sie den Bedarf nach „Vergessen“ sowie Möglichkeiten, diesen Prozess zu realisieren.

9.8 Klassen in BYOB

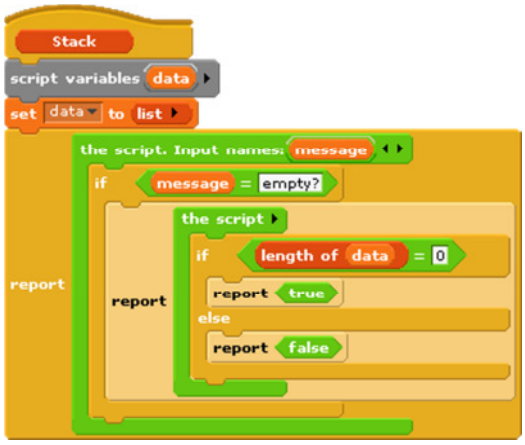
Natürlich müssen wir in BYOB nicht mit Delegation arbeiten. Wir können es, wenn wir es für nötig halten, aber wir können auch den traditionellen Weg über Klassen gehen, wenn es zur Lerngruppe und zum Problem passt. Anders als in vielen anderen Sprachen wird uns die pädagogische Entscheidung frei gestellt, sie ist nicht vom Werkzeug bestimmt.

Eine Klasse ist eine Sammlung von Daten und Methoden, die als Konstruktionsplan für Exemplare dieser Klasse dient. Wird ein solches Objekt erzeugt, dann allokiert das System den erforderlichen Speicherplatz und stellt die Operationen bereit, die mit diesen Daten arbeiten. Genau auf diese Art kann man Klassen in BYOB implementieren. Als Beispiel wollen wir den Datentyp Stapel wählen. Dieser soll eine Liste verwalten, auf die nach dem LIFO-Prinzip zugegriffen wird.

Jeder Stapel muss seinen eigenen Datenbereich haben, der bei Bedarf erzeugt wird. Daten (hier: eine Liste) „leben“ so lange im Speicher eines Computers, wie sie über Referenzen erreicht werden können. Existiert also ein Skript, das diese Daten adressiert, dann werden sie nicht gelöscht. Diesen Mechanismus nutzen wir aus: da in den Skripten einer Klasse auf die Attribute des Objekts Bezug genommen wird, betrachten wir als Objekt eine Sammlung aus Daten und Skripten, die hier ebenfalls als Daten interpretiert werden. Erhält ein Objekt die Botschaft, dass eines seiner Skripte ausgeführt werden soll, dann gibt es das entsprechende Skript zurück. Das rufende Objekt führt dieses dann ggf. mit den entsprechenden Parametern aus. Dieser Mechanismus ist aus dem oben Gesagten schon hinlänglich bekannt.

Probieren wir es also in möglichst knapper Form aus.

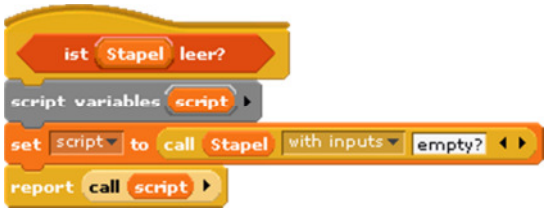
Ein Stapel enthalte eine anfangs leere Liste. Als Funktionalität wollen wir zuerst nur die Möglichkeit implementieren, anzufragen, ob der Stapel leer ist. Erhält der Stapel die Botschaft empty?, dann antwortet er mit einem Skript, das diese Frage beantworten kann.



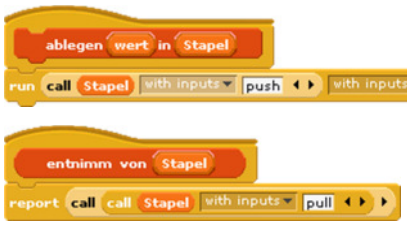
Eine Variable data erzeugen und an eine anfangs leere Liste binden.

Falls die Botschaft empty? ist, das entsprechende Skript zurück geben.

Vereinbaren wir beim rufenden Objekt eine Variable Teststapel und binden diese an einen neuen Stapel, dann brauchen wir noch eine einfache Möglichkeit, den Zustand des Stapels abzufragen: das Prädikat istLeer?. Wir verwenden hier die oben eingeführte „zweistufige“ Abfrage.



Nachdem das klappt, erweitern wir die Stack-Klasse um zwei Skripte, um Daten abzulegen (push) und wieder vom Stapel zu nehmen (pull). Benutzt werden diese Skripte wie oben gezeigt in den Methoden des rufenden Objekts. Wir benutzen jetzt die verkürzte Schreibweise:



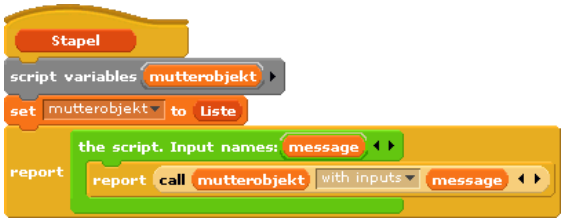
Mit diesen Operationen können wird dann recht einfach mit Stapeln hantieren.

Es ist noch einmal zu betonen, dass diese Implementierung von Klassen nur eine von vielen Möglichkeiten darstellt. Sie wird im Artikel „*No Ceiling to Scratch*“ von Brian Harvey und Jens Mönig eingeführt. Übergeben wir einem Objekt z. B. die Botschaft gleich mit allen erforderlichen Parametern in Listenform, dann kann das Objekt auch selbst die Ergebnisse ermitteln und zurückgeben.

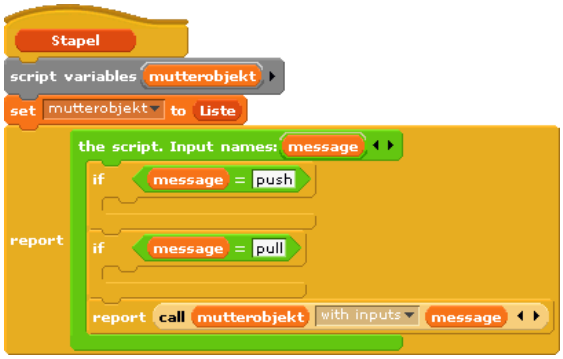
9.9 Klassen und Vererbung

Vererbung wird in dieser Version der Klassendefinition realisiert, indem die Tochterklasse ein Exemplar der Mutterklasse enthält. Erreicht ein Tochterobjekt ein Methodenaufruf, dann versucht dieses die Nachricht zu verarbeiten. Ist dafür keine Methode vorgesehen, dann reicht sie die Nachricht an die Mutter weiter. Wir wollen dieses Verfahren am Beispiel einer Stapel-Klasse demonstrieren, die von einer allgemeineren Liste-Klasse abgeleitet wird. Wir gehen davon aus, dass Listenobjekte die Nachrichten zeige (liefert Listeninhalt und Positionszeiger als Liste), leer?, gib (liefert das Listenelement an der Stelle des Positionszeigers), setze (fügt ein Element ans Ende der Liste an und setzt den Positionszeiger dorthin) und lösche (löscht das Listenelement an der Stelle des Positionszeigers) , sowie ggf. andere, hier nicht benötigte, „verstehen“.

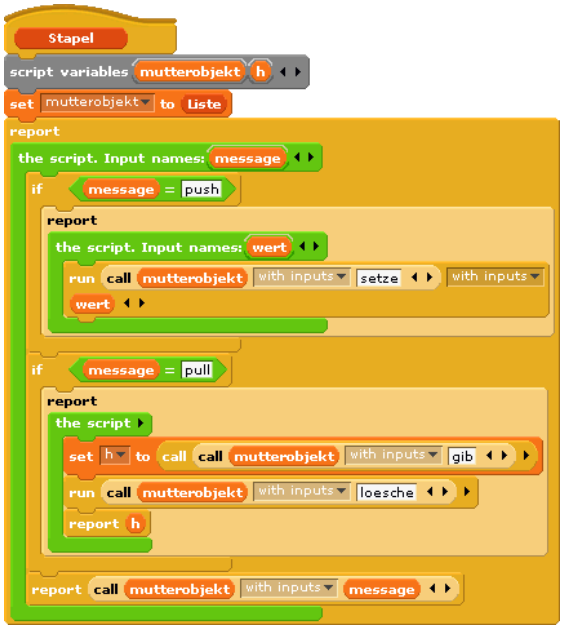
Enthält ein Objekt ein Mutterobjekt (hier: eine Liste), so kann es ganz einfach alle Nachrichten an dieses weiterleiten. Es verhält sich dann genauso wie die Mutter.



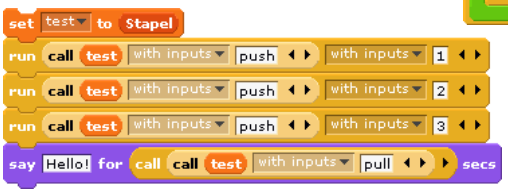
Spezielle Nachrichten für die neue Klasse verarbeiten wir vorher, in unserem Fall die Nachrichten push (legt einen Wert auf dem Stapel ab) und pull (entfernt das oberste Element des Stapels und liefert dessen Wert). Mit noch leeren Anweisungsblöcken für diese Fälle erhalten wir die nebenstehende Struktur.



Es fehlen nur noch die Operationen für die beiden speziellen Stapel-Anweisungen. Lautet die Nachricht „push“, dann liefern wir ein Skript, das einen Eingabeparameter (hier: wert) in das Mutterobjekt am Ende einfügt. Lautet die Nachricht „pull“, dann bitten wir die Mutter um das letzte Listenelement und speichern es unter dem Namen h, dann bitten wir die Mutter um Löschung des letzten Elements.



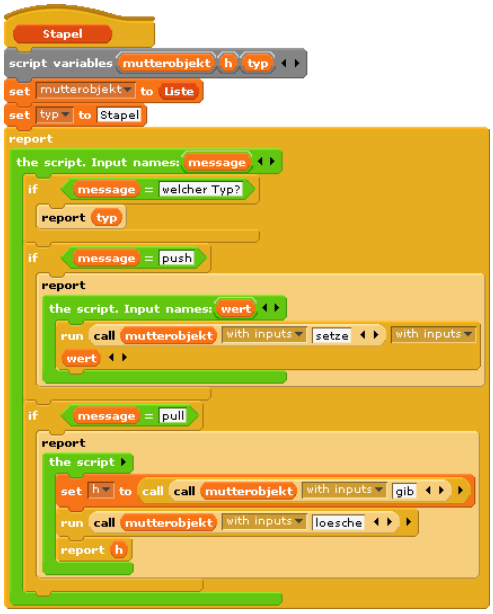
Wir benutzen diese Befehle wie schon vorher gezeigt:



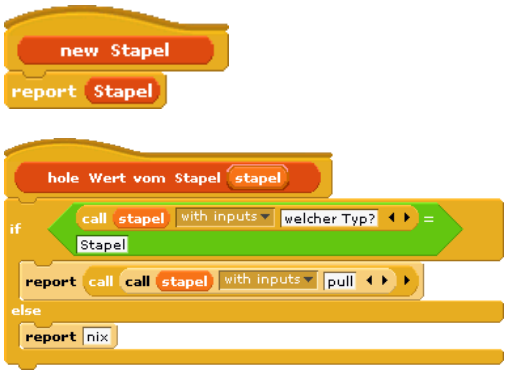
9.10 Abstrakte Datentypen

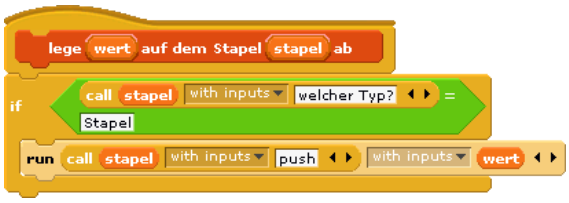
Unsere Stapel-Objekte aus dem letzten Abschnitt verhalten sich zwar wie richtige Stapel, aber wir können sie genauso als Listen benutzen – denn das sind sie als Töchter dieser Klasse auch. Außerdem ist der Zugriff auf die Operationen eigentlich nur verstehbar, wenn die Implementation als Sammlung von Daten und Skripten verstanden worden ist. Beides wollen wir vermeiden, wenn wir Abstrakte Datentypen (ADTs) verwenden. Wir definieren dann – ähnlich wie bei Anne und den Aktenschränken – eine Schnittstelle zwischen den eigentlichen Objekten und dem Benutzer. Am einfachsten ist es, einen Satz von globalen Methoden zu schreiben, der die Schnittstelle zu den Stapel-Objekten bildet.

Um Fehlzugriffe zu vermeiden, versehen wir unsere Stapel mit einem Typ und löschen den direkten Zugriff auf das Mutterobjekt. Wir erhalten das nebenstehende Skript.



Mit diesen kleinen Änderungen können wir unsere Schnittstelle gestalten:





Ein Benutzer kann damit mehrere Stapel erzeugen und wie gewohnt nutzen.

9.11 Abstrakte Datentypen „à la Java“

Gewähren wir den Zugriff auf ADTs über globale Methoden – wie eben geschildert, dann häufen sich bei Verwendung vieler solcher ADTs auch die Methoden z. B. in der Rubrik Variables. Manchen mag das stören. Wir schneidern und deshalb eine andere Zugriffsart auf ADTs, indem wir alle erforderlichen Parameter auf einmal an die Methoden des ADT übergeben. Dieser wertet sie dann selbst aus. Geschachtelte call-Aufrufe sind dann nicht mehr erforderlich: wir benötigen nur noch einen, um ein Skript im Kontext des ADTs auszuführen. Dieses kann durch zwei einheitliche Methoden geschehen, eine für Reporter und eine für Commands. Führen wir ein ADT-Sprite ein, z. B. einen ADT-Connector, dann enthält dieser die Skripte der ADTs und exportiert für jeden einen Konstruktor sowie die genannten beiden Zugriffsoperationen.

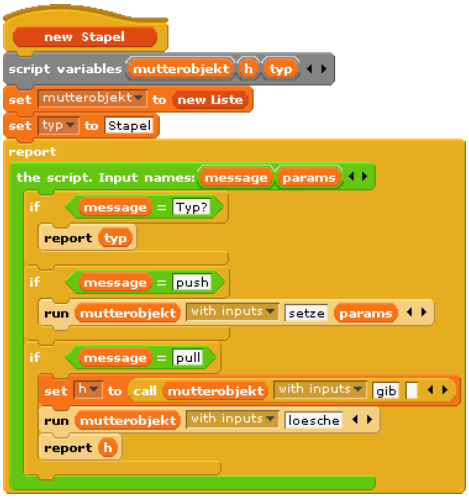
Wir demonstrieren das Vorgehen wieder an den ADTs Liste und Stapel.

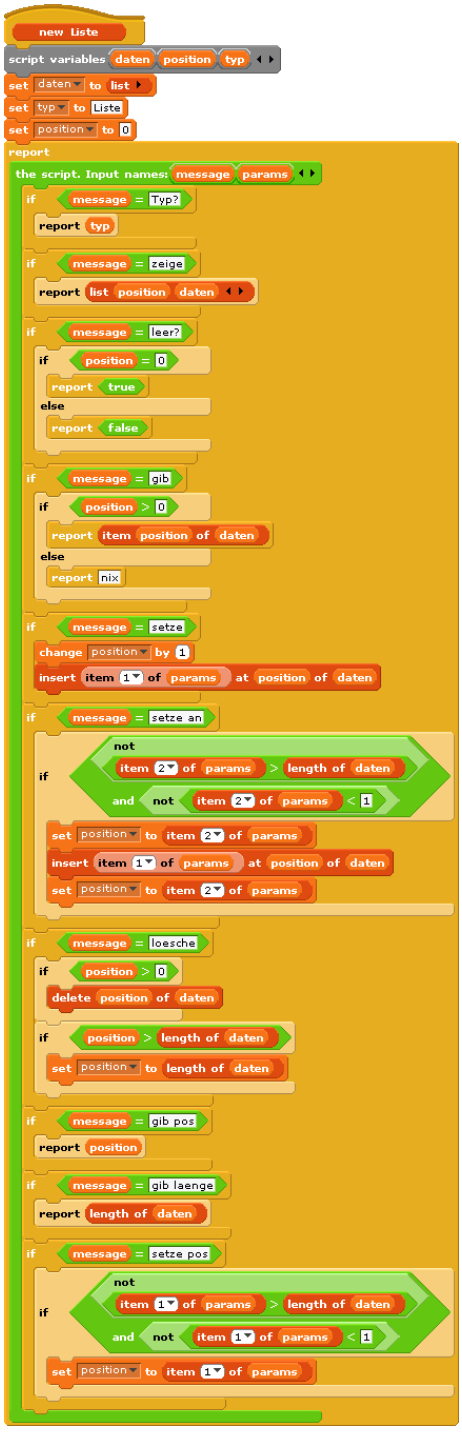
Zuerst zur Liste.

Diese „versteht“ die gleichen Befehle wie vorher, erhält aber zusammen mit dem Befehl eine (möglicherweise leere) Liste mit allen erforderlichen Parametern. Empfängt als eine Liste z. B. den Befehl „setze an“, dann ändert sie die Position des Positionszeigers auf den Wert des zweiten Elements der Parameterliste und fügt an dieser Stelle den ersten Wert der Parameterliste ein – wenn die Reihenfolge der Parameter denn so definiert wurde.

Unsere Liste ist also ein Skript, das im Kontext der oben definierten Attribute immer das gewünschte Teilskript direkt ausführt und bei Bedarf Ergebnisse zurückgibt.

Unser Stapel arbeitet entsprechend.





Wie benutzt man nun solche ADTs.

Nun, ganz einfach. Wir führen eine Art „Punktnotation“ ähnlich wie bei Java ein, nur mit einem Stern statt des Punktes. Die Syntax soll sein:

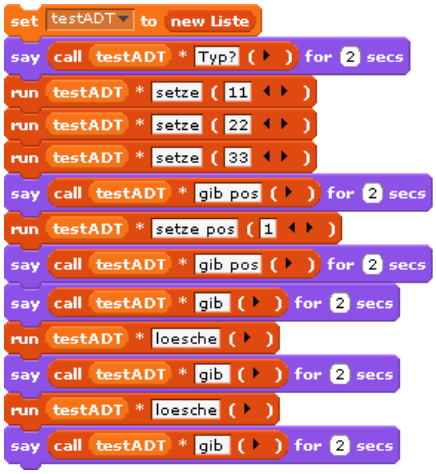
run <ADT>*<operation>(<parameter>) bzw.

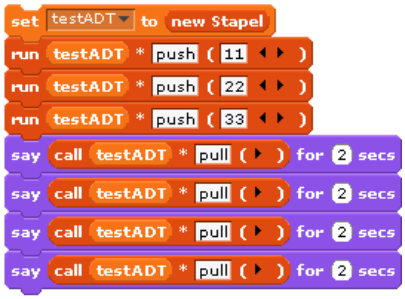
call <ADT>*<operation>(<parameter>)

Bei den ADTs handelt es sich um die Skripte, die unser ADT-Connector z. B. unter dem Namen new Liste exportiert. Diese werden an eine normale BYOB-Variablen gebunden. Damit steht der Typ dieses Parameters fest: es handelt sich um ein Inline Command (das wir hier zum ersten Mal benutzen). Die Operation geben wir als einfachen Text an (z. B. „push“) – so ist sie ja definiert. Bei den Parametern kann es sich um beliebig viele handeln, ggf. auch keine. Wir wählen deshalb den Typ multiple inputs. Den kennen wir schon z. B. von der Erzeugung neuer Listen. Mit diesen Informationen können wir unsere beiden allgemeinen Zugriffsoperationen auf ADTs dieser Art schreiben[29]:

Die neuen Kacheln sehen entsprechend aus:

Die Nutzung im Skriptbereich eines anderen Sprites demonstrieren die beiden folgenden Skripte, die mit einer Variablen testADT arbeiten.



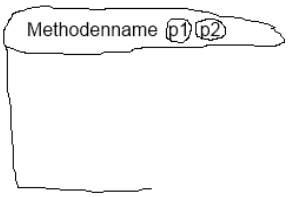
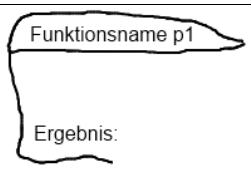
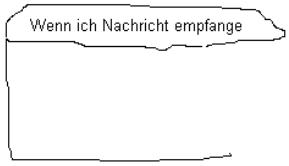
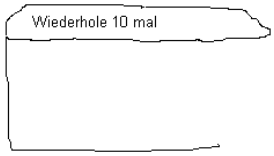
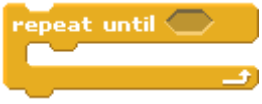
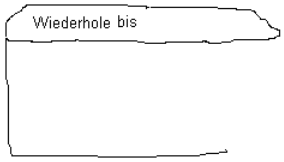


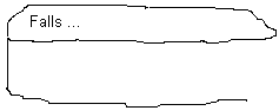
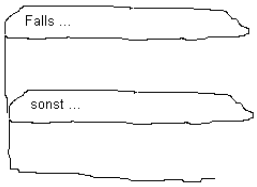
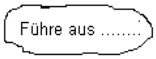
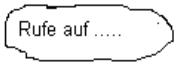
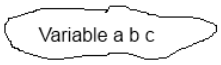
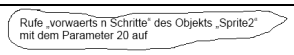
[29] Natürlich können wir die enthaltenen Anweisungen auch direkt aufrufen!

10 Zur Notation von BYOB-Programmen

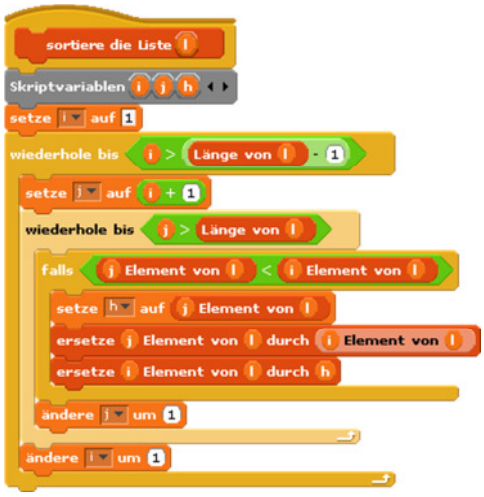
Es kommen immer wieder Einwände, dass BYOB-Programme auf Papier kaum zu notieren und Klausuren deshalb schwer zu stellen wären, denn es könnte ja wohl nicht verlangt werden, dass die Schülerinnen und Schüler dort mit Buntstiften arbeiten. Alternativ finden sich im Internet ausgefeilte Syntaxvorschläge auf diesem Gebiet. Wenn sich mir die Sinnhaftigkeit auch nicht erschließt, eine weitgehend syntaxfreie Sprache auf diesem Weg wieder mit Syntax zu behaften, und die Algorithmen eigentlich in den dafür vorgesehenen Formen (Struktogramme, UML, …) zu notieren wären, folgen jetzt doch noch zwei Vorschläge zu diesem Thema.

Zu zeigen ist also, dass in BYOB grafisch formulierte Algorithmen auch auf Papier aufschreibbar sind. Dazu müssen Methodenköpfe und algorithmische Grundstrukturen darstellbar sein. Die Schachtelung ergibt sich wie bei anderen Systemen auch durch Einrückungen und grafische Hilfsmittel.

Element	BYOB-Symbol	handschriftlich	textuell
Methodenkopf			Methodenname p1 p2
Funktionskopf			Funktionsname p1: Ergebnis
Ereignis-behandlung (Beispiel)			Wenn ich Nachricht empfange
Zählschleife			Wiederhole 10 mal
kopfgesteuerte Schleife			Wiederhole bis …

einseitige Alternative			Falls …
zweiseitige Alternative			Falls … sonst …
Evaluation eines Skripts			Führe aus …
Evaluation einer Funktion			Rufe auf …
Variablen-vereinbarung			Variable a b c
Methoden-aufruf eines anderen Objekts			Rufe „vorwaerts n Schritte“ des Objekts „Sprite2“ mit dem Parameter 20 auf

Beispiel: Sortieren einer Liste in BYOB, formal mit Einrückungen und „per Hand“ geschrieben




```
sortiere die Liste l
  Variable i j k
  setze i auf 1
  wiederhole bis i > Länge von l – 1
    setze j auf i+1
    wiederhole bis j > Länge von l
      falls j-tes Element von l < i-tes Element von l
        setze h auf j-tes Element von l
        ersetze j-tes Element von l durch
                                  i-tes Element von l
        ersetze i-tes Element von l durch h
      ändere j um 1
  ändere i um 1
```