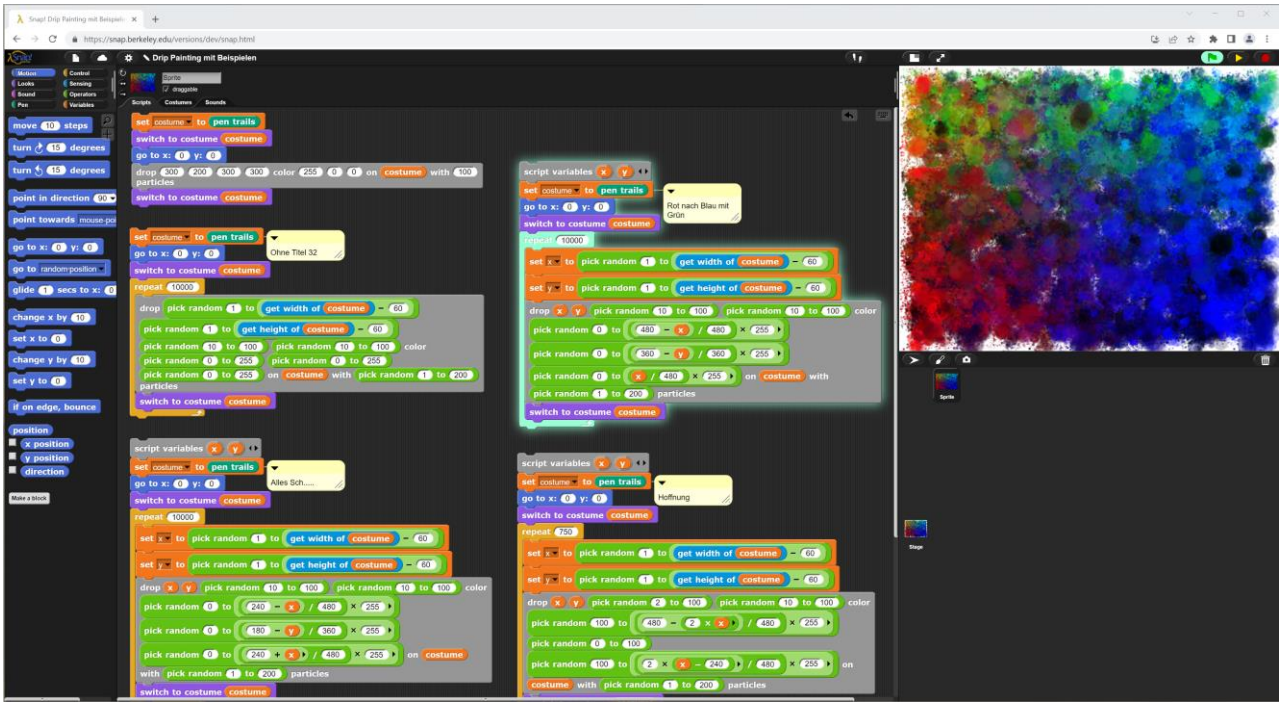
Eckart Modrow

Informatik mit Snap!

– *Snap!* in Beispielen –

Version 2



emodrow@informatik.uni-goettingen.de

Prof. Dr. Modrow, Eckart:

Informatik mit *Snap!*

- *Snap!* in Beispielen -

Version 2

Wenn Sie mit diesem Buch zufrieden sind und ihre Anerkennung in Form einer Spende zeigen möchten, dann können Sie das auf folgendem PayPal-Konto tun:

emodrow@emu-online.d
Verwendungszweck: Snap!-

Vorwort

Das vorliegende Skript stellt, ähnlich wie der Vorgänger „*Informatik mit Snap!*“[1], anhand einer Sammlung von Programmierbeispielen die Möglichkeiten der grafischen Sprache *Snap!* dar. Es ersetzt kein Lehrbuch, das informatische Inhalte vermittelt, sondern zeigt, wie informatische Methoden mithilfe von *Snap!* angewandt werden. In dieser zweiten Version sind einige Überlegungen zum Informatikunterricht, besonders zum Objektbegriff und zum Verhältnis von Information, Daten und Visualisierung, vorangestellt. Beispiele, die deren Konsequenzen erläutern, findet man im Folgenden.

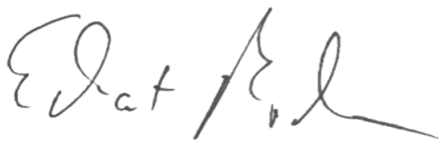
Snap! in der vorliegenden Version 8.0.0 bildet den nächsten Schritt in der Entwicklung der grafischen Werkzeuge. Die aktuelle Version wurde u. a. um Möglichkeiten im Bereich der Objektorientierten Programmierung (OOP), der Listenoperationen und mehrerer Bühnen erweitert und genügt damit allen Anforderungen bis zum Abitur und weit darüber hinaus. Da auch beim Arbeitstempo drastische Verbesserungen erreicht wurden und Bibliotheken für unterschiedliche Bereiche, z. B. beim Pixelzugriff, im Audiobereich oder bei der Nutzung externer Ressourcen bereitstehen oder leicht neu entwickelt werden können, bestehen kaum noch Einschränkungen in den Anwendungsgebieten. Wenn es sein muss, kann man nach wie vor JavaScript-Funktionen z. B. für zeitkritische Operationen oder Erweiterungen innerhalb von *Snap!* benutzen. Die Bibliotheken enthalten zahlreiche Beispiele dafür.[2]

Die Auswahl der Probleme in den folgenden Kapiteln ist relativ konservativ, lehnt sich teilweise eng an bestehenden Informatikunterricht an, geht aber auch darüber hinaus. Das ist Absicht. Ich hoffe, damit einerseits die unterrichtenden Kolleginnen und Kollegen vom traditionellen Unterricht „abzuholen“, andererseits Kontexte zu liefern, die den zu erwerbenden informatischen Inhalten aus Sicht der Lernenden einen Sinn geben. Auf diesem Weg sollte sich ein sehr an Kreativität, aber auch an der Vermittlung informatischer Konzepte ausgerichteter Unterricht ergeben. Die Beispiele beschreiben detailliert den Umgang mit *Snap!* unter verschiedenen Aspekten. Nach ein paar Überlegungen zur Didaktik in diesem Bereich folgt ein Einführungskapitel, das die Arbeit mit *Snap!* „auf die Schnelle“ erläutert. Dann werden in den ersten Kapiteln die Möglichkeiten der Sprache illustriert. Es folgen auch Abschnitte ohne direkten Anwendungsbezug. Dieser Kompromiss ist dem Platzbedarf geschuldet, weil erweiterte Konzepte eigentlich auch erweiterte Problemstellungen erfordern. Die Beispiele sind nicht hierarchisch geordnet, auch im zweiten Teil finden sich eher einfache. Am Ende des Skriptes befinden sich Übersichten über die in den Beispielen verwendeten Methoden sowie ein Index.

Ich bedanke mich sehr bei Jens Mönig für seine Unterstützung – und für die Ergebnisse seiner Arbeit. Die Lernenden werden es ihm danken!

Ansonsten wünsche ich viel Freude bei der Arbeit mit *Snap!*

Göttingen, am 1.8.2022



[1] E. Modrow, Informatik mit Snap, https://www.emu-online.de/InformatikMitSnap.pdf

[2] Auf *SciSnap!2* wird genauer eingegangen in http://emu-online.de/ProgrammierenMitSciSnap2.pdf

Inhalt

1 Didaktische Anmerkungen

1.1 Daten, Informationen, Geschichten und Visualisierungen[3]

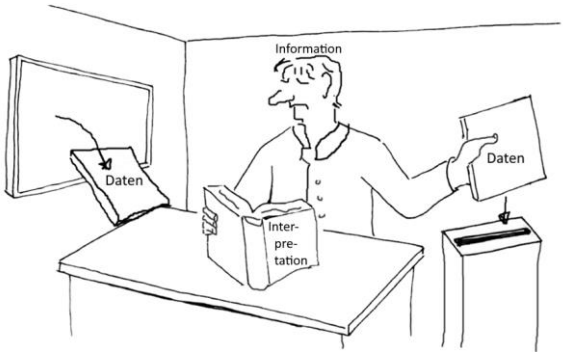
Zum Kern der prozessbezogenen Kompetenzen der Schulinformatik gehören das *Modellieren* und *Implementieren* sowie das *Begründen* und *Bewerten*. Für den Unterricht ist deren Zusammenhang entscheidend: einerseits sollen die Lernenden selbstständig Problemlösungen erstellen, wozu sie fachliche Grundlagen erwerben und natürlich auch eine Schulung im Werkzeuggebrauch benötigen, andererseits soll die Thematik den Diskurs über gesellschaftliche und politische Fragenstellungen auf der Basis der erworbenen Fachkompetenz ermöglichen. Das Verhältnis der drei Bereiche *Werkzeuggebrauch*, *Fachfragen* und *Auswirkungen* bestimmt den Rahmen für allgemeinbildenden Unterricht. Oder etwas schärfer formuliert:

> *Welchen Anteil darf die Werkzeugschulung, also das Erlernen des Umgangs mit der benutzten Programmiersprache incl. deren Entwicklungsumgebung, im Unterricht haben, damit genügend Zeit für selbstständiges Problemlösen der Lernenden sowie die Reflexion der Ergebnisse bleibt?*

Ohne diese Zeit hat das Fach an allgemeinbildenden Schulen eigentlich nichts zu suchen. Im Folgenden untersuchen wir etwas genauer die im deutschsprachigen Raum verbreitete *informationsorientierte Informatikdidaktik*, die darin benutzte Begrifflichkeit und die Auswirkungen auf die Wahl des Werkzeugs und seine Nutzung.

Die deutsche Gesellschaft für Informatik (GI) schreibt zu den genannten Kompetenzen:

„Der Prozess der Modellierung ist nicht nur Lerninhalt, sondern auch durchgängige Methode des Informatikunterrichts, wobei aber auch die Implementierung unverzichtbar ist, um das Ergebnis der Modellbildung erlebbar zu machen. Begründen und Bewerten fördern die Kommunikations- und Argumentationsfähigkeit des Lernenden, ohne diesen Bereich ist der Umgang mit Informatiksystemen nur intuitiv oder spielerisch und häufig durch Einflüsse aus Medien bestimmt.“

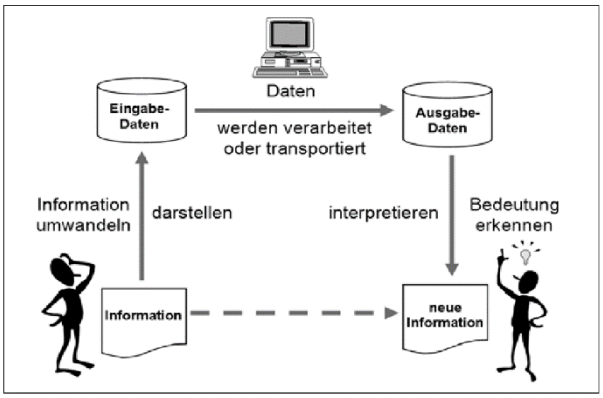


Als Inhalte nennt die GI für die Mittelstufe u. a. den *Zusammenhang von Information* und *Daten*, verschiedene *Darstellungsformen* und *Operationen auf Daten* und deren *Interpretation* in Bezug auf die dargestellte Information. In der Oberstufe soll u. a. zwischen *Zeichen*, *Daten* und *Information* sowie zwischen *Syntax* und *Semantik* unterschieden und *Information als Daten* mit Datentypen und in Datenstrukturen abgebildet werden. Die aktuellen Curricula übernehmen diese Vorgaben weitgehend. Neben den Inhalten sind für Lehrende die Beispielaufgaben besonders interessant, weil sich aus ihnen besonders gut eine Vorstellung von dem intendierten Unterricht gewinnen lässt. Im betrachteten Gebiet finden sich traditionell behandelte Themen aus dem Bereich der Datenstrukturen und Datenbanken, aber praktisch nichts über Informationen. Dieser Begriff tritt meist nur innerhalb von Wortkombinationen (*Informations*technik, *Informations*gesellschaft, …) auf, und er wird widersprüchlich verwandt. Wenn z. B. Information als *„die Semantik einer Aussage, Beschreibung, Anweisung, Mitteilung oder Nachricht“* [4] definiert wird, dann erschließt sich mir nicht

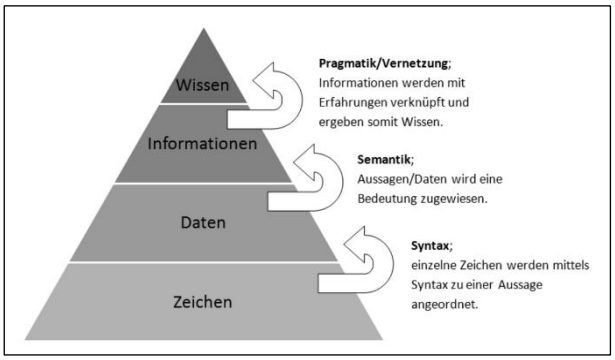
[3] weitgehend nach Modrow, E., (2017). Ist der Informationsbegriff für die Schulinformatik hilfreich? LOG IN: Vol. 37, No. 1. Berlin: LOG IN Verlag. (S. 38-43).

[4] https://kultusministerium.hessen.de/schule/kerncurricula/gymnasiale-oberstufe/informatik

ganz, wie diese Semantik *„durch Maschinen automatisch verarbeitet“*[5] werden soll. Aus einem nicht genügend scharf definierten Begriff lässt sich anscheinend nicht so leicht Unterricht ableiten, auch wenn er prominent in den Kompetenzbereichen benutzt wird. Es ist also lohnend, sich etwas eingehender mit der Bedeutung von Information zu beschäftigen.



Der Informationsbegriff wird in der informationszentrierten Informatikdidaktik meist anhand des nebenstehenden Schemas erläutert.[6] Leitet man daraus Inhaltsbereiche ab, dann kommt man sehr schnell z. B. zur automatischen Verarbeitung und Vernetzung von *Repräsentationen*, also zu Daten. Aus der informationszentrierten Didaktik wird ebenso schnell eine datenzentrierte, wenn es um konkreten Unterricht geht. Aus dem Diagramm wird deutlich, dass der in der Informatikdidaktik benutzte Informationsbegriff weder mit dem Shannonschen der Informationstheorie noch mit der umgangssprachlichen Gleichsetzung von Information und Daten viel zu tun hat. Die Ebene der Informationen ist darin kaum mit informatischen Fachinhalten verknüpft, sodass eine Umsetzung in Unterricht schwerfällt bzw. inhaltliche Brüche erfordert.



Wir benötigen deshalb präzise und miteinander kompatible Definitionen für die benutzten Begriffe. Hilfreich scheint mir dafür die *Wissenspyramide*[7] zu sein, die neben *Daten* und *Informationen* auch noch die Ebenen des *Wissens* und der *Zeichen* enthält. Als Ausgangspunkt wählen wir die Definition von Wissen aus der Wikipedia[8]:

> ***Wissen*** *wird […] als ein für Personen oder Gruppen verfügbarer Bestand von Fakten, Theorien und Regeln verstanden, die sich durch den größtmöglichen Grad an Gewissheit auszeichnen, so dass von ihrer Gültigkeit bzw. Wahrheit ausgegangen wird.*

Wissen ist somit an Personen gebunden und kann folgerichtig z. B. innerhalb heutiger Maschinen nicht existieren. Dort finden wir Daten. Da Wissen nicht vollständig und sogar falsch sein kann, ergeben sich Lücken in der Gewissheit, die durch Informationen geschlossen oder verringert werden können[9].

> ***Information*** *ist die Teilmenge von Wissen, die von einer bestimmten Person oder Gruppe in einer konkreten Situation benötigt wird und häufig nicht explizit vorhanden ist.*

Diese Definition entspricht in etwa der aus den GI-Bildungsstandards, *„Information ist der kontextbezogene Bedeutungsgehalt einer Aussage, Beschreibung, Anweisung, Mitteilung oder Nachricht.“*, allerdings auf das durch die Information geänderte Wissen bezogen. Information ist ebenfalls an Personen gebunden, die den Bedeutungsgehalt der Daten erkennen und bewerten. Sie ist zeit- und situationsabhängig. Erhält eine Person z. B. eine Nachricht zweimal, dann ist der Informationsinhalt beim zweiten Mal sehr viel geringer, denn die Wissenslücke wurde schon von der ersten Information geschlossen. Informationen hängen einerseits

[5] http://www.schulentwicklung.nrw.de/lehrplaene/upload/klp_SII/if/KLP_GOSt_Informatik.pdf
[6] http://www.informatikstandards.de/index.htm
[7] https://derwirtschaftsinformatiker.de/2012/09/12/it-management/wissenspyramide-wiki/
[8] https://de.wikipedia.org/wiki/Wissen
[9] https://de.wikipedia.org/wiki/Information

von den zu ihrer Übermittlung benutzten Daten ab, aber andererseits auch vom Zustand des Empfängers. Dieser baut pragmatisch Informationen in sein bestehendes Wissen ein, vernetzt sie mit diesem – oder auch nicht. Bis hierher gibt es keine Probleme: Informationen befinden sich im Kopf, Daten im Rechner. Der Informationsbegriff hat auf der maschinellen Ebene, der wir uns jetzt zuwenden, nichts zu suchen.

> *Daten werden durch **Zeichen** des gewählten Zeichensatzes repräsentiert, den wir hier als Code auffassen können. Die **Syntax** dieser Repräsentation beschreibt die Struktur dieser Darstellung.*

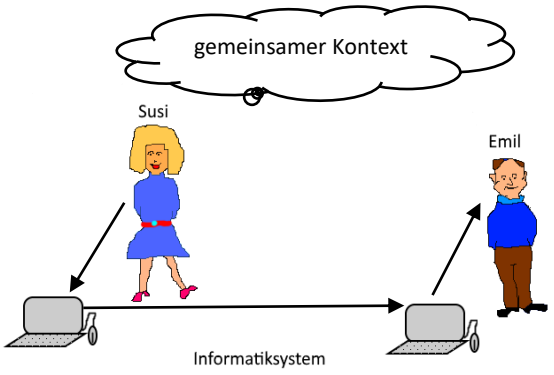
Der oben genannte Informationsbegriff ist personenbezogen. Informationen können also nicht ohne die interpretierende Person gesehen werden, z. B. weil dieselben Daten für unterschiedliche Personen ganz unterschiedliche Informationen darstellen können. Ohne ihren ***Kontext*** verlieren Daten die Eigenschaft, Information zu sein. Sie werden auf das reduziert, was sie ohne Bedeutung sind: eben Daten. Im Modell der Wissenspyramide sind die Verhältnisse klar: der Empfänger interpretiert die empfangenen Daten und versucht, sich ihre Semantik zu erschließen. Dieser Schritt erfolgt vor der Vernetzung mit seinem bestehenden Wissen und weitgehend unabhängig von dieser. Die Interpretation ist vom Empfänger und seinem Zustand abhängig, sie kann nicht ausschließlich aufgrund der Daten erfolgen. Nach der Interpretation entscheidet der Empfänger, ob die Bedeutung der Daten für ihn eine Information darstellt.

Wir sollten m. E. darauf verzichten, den Datenbegriff wie oben geschehen, in das informationszentrierte Schema zu pressen. Daten sind eine Kategorie an sich, gebunden an eine physische Repräsentation. Misst z. B. eine Meeressonde Temperaturwerte, speichert diese und geht anschließend verloren, dann sind die physisch repräsentierten Messwerte als Daten existent, auch wenn sie bedauerlicherweise nie zu einer Information werden. Speichert ein Betriebssystem den Systemzustand in Protokolldateien, dann sind diese Daten vorhanden, auch wenn sie nie von Menschen ausgewertet werden. Die Definition von Daten als Repräsentationen von Informationen verwechselt den oben genannten Informationsbegriff mit dem umgangssprachlichen und führt zu der unschönen Situation, dass der Bedeutung der Information Genüge getan zu sein scheint, wenn Daten und ihre Strukturen betrachtet werden. Das ist es aber nicht.

Unsere Untersuchung hat ein einfaches Ergebnis: Die beiden untersten Ebenen der Wissenspyramide sind der Fachwissenschaft Informatik zugänglich. Sie sind mit den tradierten Inhaltsbereichen verknüpft. Die beiden oberen sind zumindest teilweise intrapersonal, gehen über die reine Fachwissenschaft hinaus. Wie der Bereich „Informatik und Gesellschaft“ beziehen sie sich auf die Bedeutung der Informatiksysteme, diesmal nicht so sehr politisch und sozial gesehen, sondern auf die persönliche Betroffenheit bezogen. Der Informationsbegriff gehört zum allgemeinbildenden Beitrag der Schulinformatik. Dieser wird nicht erreicht, wenn die Behandlung von datenbezogenen Themen mit informationsbezogenen gleichgesetzt wird.

Wir wollen die Konsequenzen unserer Überlegungen in vier Situationen durchspielen. Dafür benennen wir die beiden Akteure des oben gezeigten Informationsübertragungsschemas als *Susi* (Sender) und *Emil* (Empfänger) und reduzieren die Beschriftung im Schema.

1. Fall: Susi sendet die Nachricht *„Bin angekommen!“* an Emil.



Kommunikation in gegebenem Kontext

Die Nachricht kann für Emil nur eine Bedeutung haben, wenn Susi und er sich über ihren Sinn im Klaren sind. Weiß also Emil, dass Susi entweder auf dem Weg nach Hannover oder zu sich selbst ist, dann kann er die Nachricht interpretieren, sogar inklusive von Subtexten wie dem fehlenden „gut“, die einige Komplikationen erwarten lassen. Susi wiederum weiß, dass Emil auf ihre Nachricht wartet und

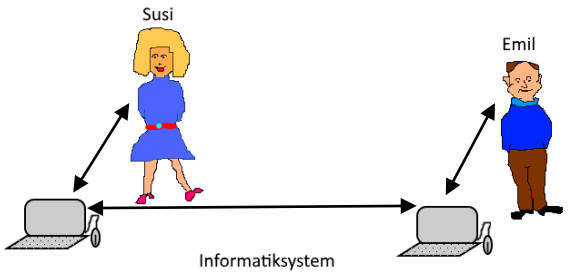
sie in ihrer Kürze verstehen wird. Sie kann ihre Information durch entsprechende Daten ausdrücken. Susi und Emil handeln innerhalb eines gemeinsamen Kontextes, der es gestattet, die Nachricht zu interpretieren. Ohne diesen Kontext ist das nicht möglich, und deshalb sollte der Kontext auch ins Schema aufgenommen werden. Dort darf er aber nicht verbleiben, denn relevant sind natürlich die unterrichtlichen Konsequenzen, nicht die schematischen. Im Unterricht kann solch ein Kontext gut durch ***Geschichten*** realisiert werden, so wie wir es auch gerade tun. Es geht also nicht nur um geeignete Datenstrukturen und Protokolle, sondern ebenso um die ***Visualisierung*** des Geschehens, die Anbindung der Fachthemen, die sich aus einem Problem ergeben, an die Akteure der Geschichte, seien es nun die Bewohner eines Bauernhofs, die Beziehungskiste zwischen Susi und Emil oder die Elemente einer Simulation. Die Manipulation der darin vorkommenden Daten sollte ebenso bei Bedarf möglich sein, um ohne Aufwand das Geschehen beobachten und die Ergebnisse kontrollieren zu können, denn was man sieht, braucht man meist nicht gesondert zu erklären. Beides, die Visualisierung des Kontextes und der Daten, sollte eine schulgeeignete Entwicklungsumgebung einfach ermöglichen.

Die Rolle des Informatiksystems ist in diesem ersten Fall völlig nebensächlich, klar vom Informationsaustausch getrennt. Susi hätte auch laut rufen, eine Postkarte senden, die Nachricht trommeln oder per Brieftaube transportieren lassen können. Und umgekehrt ist die Fähigkeit des Informatiksystems, Texte geeignet zu kodieren, die Zeichen zu transportieren und wieder darstellen zu können, völlig unabhängig vom Informationstransport. Die Interpretationsaufgabe des Systems besteht darin, die Zeichen so zu kennzeichnen, dass sie als Text erkannt und durch ein geeignetes Teilsystem dargestellt werden können. Diese Aufgabe erfolgt automatisch z. B. durch Kennzeichnung der Datenpakete oder der Datei aufgrund der vereinbarten Syntax. Mit Verständnis hat das nichts zu tun.

Insgesamt ist das Beispiel, das durch das Grundschema nahegelegt wird, unter informatischen Gesichtspunkten unergiebig. Man sollte es m. E. nur benutzen, wenn der eher allgemeine Informationsaspekt im Unterricht besonders hervorgehoben werden soll.

2. Fall: Susi sendet die Nachricht *„meistens nachmittags“* an Emil.

In diesem Fall soll der gemeinsame, in vielen Krisen gestählte Kontext nicht vorhanden sein, weil Susi und Emil mehr oder weniger zufällige Kommunikationspartner im Netz sind. Da Susi ohne diesen Kontext keine angemessenen Daten zusammenstellen und übermitteln kann, muss Emil den Kontext zuerst herstellen. Der Kommunikationsprozess muss deshalb von ihm gestartet werden, indem er eine entsprechende Frage an Susi stellt. Die Frage wird von Susi so interpretiert, dass sie die gewünschte Information identifizieren und in Daten umsetzen kann. Die empfangenen Daten muss Emil wiederum als Antwort auf seine Frage interpretieren und als die gesuchte Information bewerten. Das wird im Schema durch Doppelpfeile dargestellt. Dabei kann auf beiden Seiten viel schiefgehen. Susi kann die Frage falsch verstehen, wenn diese nicht völlig eindeutig formuliert ist. Sie kann also falsche Informationen von Emil erhalten und entsprechend falsche Antworten erzeugen, die von Emil wiederum falsch verstanden werden können.



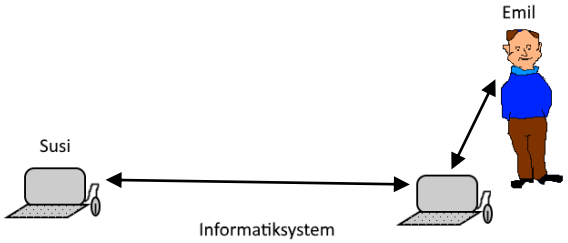
Kommunikation mit offener Fragestellung

Auch in diesem Fall passieren die interessanten Dinge in den Köpfen der Beteiligten. Wir könnten die Bedeutung der nichtverbalen Kommunikation diskutieren und die Rolle der Emoticons thematisieren, das Textverständnis in unterschiedlichen sozialen oder kulturellen Kontexten untersuchen oder den Bedarf an Bildtelefonie. All das sind wichtige und diskussionswürdige Schulthemen. Ihnen gemeinsam ist aber, dass

wir uns ihnen weder über Kenntnisse der Netzprotokolle noch der benutzten Datenstrukturen nähern. Die informatischen Fachthemen sind für die hier diskutierte Rolle der Information irrelevant.

3. Fall: Susi sendet die Nachricht *„Berlin, Bern, Bukarest“* an Emil.

In diesem Fall hat Emil seine Frage so präzise gestellt, dass Susi sie eindeutig auswerten kann. Eine Interpretation und somit ein für Verständnis geeigneter Kontext ist nicht erforderlich. Damit entfällt aber auch die Rolle von Susi als Person. Sie kann durch einen Computer ersetzt werden, der die Frage beantwortet, solange einige Syntaxregeln eingehalten werden. Emil kann z. B. fragen:



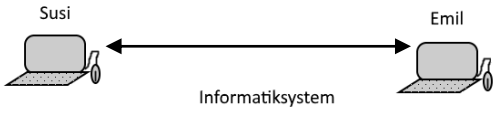
Kommunikation mit eindeutiger Fragestellung

SELECT name FROM staedte WHERE istHauptstadt = „ja“ AND name like „B%“ LIMIT 3;

Die Information ist in diesem Fall sehr einseitig verteilt. Emil weiß, welche Information er benötigt. Er beschreibt die Daten, die zum Schließen der Wissenslücke erforderlich sind und ruft diese von einem Informatiksystem ab. Weder auf dem Weg von Emil zum System noch innerhalb des Systems ist auch nur ein Ansatz von Information zu finden. Diese entsteht erst in Emils Kopf, nachdem er Susis Antwort erhalten hat.

Da dieser dritte Fall sehr direkt der Kommunikation in und mit Informatiksystemen entspricht, ist seine Analyse für die Lernenden wichtig. Egal, ob sie digitale Assistenten benutzen, Datenbanken befragen oder Suchmaschinen benutzen: von ihnen wird eine eindeutige Beschreibung der zur Fragenbeantwortung erforderlichen Daten erwartet – ob es ihnen passt oder nicht. Die Systeme spiegeln zwar Verständnis vor oder es wird ihnen von den Benutzern zugesprochen, aber sie besitzen es nicht. Das Bewusstsein dafür verhindert die Überbewertung der erhaltenen Antworten und die Unterbewertung der Verantwortung des Benutzers für seine Fragestellung. Je mehr die Rolle der Kommunikationspartner verwischt wird, desto unklarer wird die Bewertung der Ergebnisse.

4. Fall: Emil überträgt seine Aufgaben an ein Programm und geht schwimmen.



Datenaustausch ohne menschliche Partner

Nachdem Susi schon durch einen Algorithmus, in diesem Fall durch einen SQL-Server, ersetzt wurde, könnte ja auch Emil auf die Idee kommen, dass seine Aufgaben besser und schneller von einem Algorithmus wahrgenommen werden können. Er behauptet, dass er seine Interpretation von Susis Daten ausreichend präzise durch ein Programm beschreiben kann, das den Daten Informationen entnimmt und auch gleich ggf. erforderliche Aktionen veranlasst. Stimmt das? Wir wählen als Beispiel den Hochfrequenzhandel im Bankensystem. Susi übermittelt die aktuellen Kurse an ihrem Börsenplatz, Emil bewertet die Unterschiede zu seiner Börse und veranlasst entsprechende Kauf- oder Verkaufsanweisungen.

Da Emil, jetzt als Maschine, über kein Wissen verfügt, können auch keine Wissenslücken bei ihm geschlossen werden. Um Information im definierten Sinne kann es sich also nicht handeln. Der Algorithmus Emil ist zwar aus dem Wissen der Person Emil über die Abläufe im Börsenhandel entstanden, er stellt dieses Wissen aber nicht vollständig dar, und vor allem, er vernetzt es nicht mit Emils restlichem Wissen. Die Lücken in diesem Wissen, die für konkrete Reaktionen im Börsenhandel geschlossen werden müssen, erfordern die aktuellen Börsenwerte. Der Algorithmus verfügt dafür über Variable, also Leerstellen, die von Susi aktuali-

siert werden. In Abhängigkeit von diesen Werten durchläuft Emil seine Anweisungsfolgen in unterschiedlicher Reihenfolge und löst die entsprechenden Aktionen aus. Dafür ist keinerlei Interpretation erforderlich. Es handelt sich um einen reinen Automatisierungsvorgang.

Wir können aus den vier betrachteten Fällen einiges lernen. Die ersten beiden zeigen, dass menschliche Kommunikation problematisch sein kann, und zwar unabhängig vom benutzen Medium. Dieses erhält seine Bedeutung daraus, dass es Kommunikation ermöglicht und aus seiner Verbreitung. Für die dabei auftauchenden technischen Fragen der Datenverarbeitung ist der Informationsbegriff irrelevant.

Interessanter sind die beiden anderen Fälle. Der dritte beschreibt ganz gut die Rollen von Benutzer und Informatiksystem bei der Informationsbeschaffung. Die Intelligenz liegt dabei vollständig beim Benutzer. Dieser beschreibt die zur Erzeugung der gesuchten Information erforderlichen Daten und ist damit auch für diese Beschreibung verantwortlich. Ist die Beschreibung unpräzise, dann erhält er entsprechende Antworten. Befragt der Fragende wie in Fall 2 einen menschlichen Experten, dann muss sich dieser die gesuchte Information aus dem Kontext erschließen und mithilfe des Informatiksystems die zur Beantwortung erforderlichen Daten zusammenstellen und übermitteln. Er übernimmt dann auch die Verantwortung für deren Relevanz. In Fall 3 steigen die Anforderungen an den Frager erheblich, denn er muss jetzt Experte sein. Ausreden gibt es nicht mehr. Seine Frage wird immer ausgewertet, z. B. über statistische Zusammenhänge oder indem Übereinstimmungen mit dem Fragetext wörtlich im Netz gesucht werden, sie wird aber nicht verstanden. Um die anfallenden Daten überhaupt sortieren zu können, muss das System den fehlenden Kontext ergänzen, z. B. durch Auswertung vergangener Fragen oder ähnlicher Fragen anderer. Die Gefahr, dass so z. B. „Echokammern“ entstehen, die Daten immer der gleichen Tendenz erzeugen, wird als aktuelles, die Demokratie gefährdendes Problem diskutiert.

Der Informationsgesichtspunkt führt in diesem Szenario auf die Frage, was der Benutzer wissen muss, um angemessene Fragen stellen zu können; wissen sowohl vom Thema der Fragestellung wie von der Funktionsweise des benutzen Systems. Die traditionellen Fachthemen der Schulinformatik werden damit um einen Aspekt erweitert, der geeignet ist, die Relevanz eben dieser Fachinhalte vor dem Hintergrund des Lebens in einer durch Informatiksysteme geprägten Gesellschaft zu bewerten. Eine so verstandene informationszentrierte Didaktik erfordert die Entwicklung neuer Unterrichtskomponenten, um das Fach in Richtung von aktuellem allgemeinbildendem Unterricht weiterzuentwickeln. Sie verzahnt die Fachinhalte mit deren gesellschaftlicher Bedeutung. Um das mit vertretbarem Aufwand leisten zu können, erfordert es Werkzeuge, die einerseits den Zeitbedarf für die Werkzeugschulung klein halten, also Zeit für anderes freiräumen, und andererseits neben einer angemessenen Berücksichtigung der Fachthemen auch dem Kontext in Form von Geschichten Raum geben.

Der vierte Fall beschreibt die Übertragung von menschlichen Aufgaben an Informatiksysteme. Menschen können aus ihrem Wissen und ihrer Erfahrung heraus beschreiben, wie in unterschiedlichen Situationen reagiert werden sollte. Die Methoden des maschinellen Lernens überführen dann Beschreibungen dieses Wissens in geeignete (Daten-)Strukturen. Im Rahmen dieser Strukturen reagieren die automatisierten Systeme dann in den Menschen vergleichbarer Weise, meist sogar schneller und zuverlässiger. Was aber passiert, wenn die Beschreibung unvollständig ist oder neue Situationen auftreten? Da die ausgewerteten Daten ihren Datencharakter im gesamten Prozess beibehalten, also nie zu Informationen werden, wird ihre Semantik auch nie erschlossen. Bedeuten sie etwas anderes als eigentlich vorgesehen, dann versteht niemand diese Bedeutungsänderung, weil sie nicht mit vorhandenem Wissen aus vielleicht ganz anderen Gebieten verknüpft werden kann. (Nebenbei: auch die Nutzung Neuronaler Netze ändert an dieser Bewertung nichts.) Die klare Trennung von Daten und Informationen ermöglicht in diesen Fällen, z. B. die Verantwortlichkeit für die Konsequenzen der Automatisierung zu diskutieren (etwa beim autonomen Fahren) und die ethischen Grenzen auszuloten, etwa bei der Auswahl der Trainingsdaten. Der Informationsaspekt schafft

Klarheit bei der Argumentation und verhindert, dass gesellschaftlich relevante Fragen durch den Rückzug auf Fachinhalte vernebelt werden. Sie ermöglicht den politischen Diskurs über die Stellung der Informatiksysteme.

Die informationszentrierte Didaktik hat dazu geführt, dass der Informationsbegriff zumindest im deutschsprachigen Raum etwas inflationär in fast allen Gebieten der Schulinformatik benutzt wird. Er verliert dabei an Schärfe und vor allem die Funktion, Orientierung bei der Planung von Unterricht zu geben. Den traditionellen Inhaltbereichen wie z. B. dem der Daten und Datenstrukturen schadet es zwar nicht, dass ihnen jetzt zusätzlich der Anspruch zugeschrieben wird, ebenfalls dem Informationsaspekt Rechnung zu tragen. Es leidet aber Chance, den Informatikunterricht in Richtung seiner allgemeinbildenden Funktion zu akzentuieren. Reduzieren wir den Informationsbegriff dagegen auf seine ursprüngliche Bedeutung, dann erweitern wir den Themenkanon der Schulinformatik um gesellschaftlich relevante Aspekte, die sich direkt auf die Planung der Curricula auswirken können.

Als Beispiel betrachten wir den Begriff der „Wissensgesellschaft“, aus dem teilweise gefolgert wird, dass Wissen nicht mehr erworben werden muss, wenn es „im Netz“ allen zur Verfügung steht. Anhand unserer Überlegungen sieht man sofort, dass es so einfach nun doch nicht geht. Im Netz finden wir kein Wissen, sondern Daten. Es ergeben sich stattdessen einige Fragen, die geklärt werden müssen, bevor es mit der Informationsgewinnung in der Wissensgesellschaft so richtig klappen kann:

- Über welches Grundgerüst von Wissen müssen die Lernenden verfügen, um ihre Wissenslücken überhaupt erkennen zu können?
- Welche Kompetenzen müssen die Lernenden erwerben, um die zum Schließen der Wissenslücken erforderlichen Daten präzise beschreiben zu können? Können sie überhaupt beschreiben, was sie nicht wissen?
- Welches Wissen über die Daten liefernde Informatiksysteme müssen die Lernenden erwerben?
- Wie lernen die Lernenden, die Relevanz der gelieferten Daten bezogen auf ihre Fragestellung einzuschätzen?
- Was passiert, wenn die Antworten „gefärbt“, z. B. auf die Fragenden abgestimmt werden?
- Welche Daten gewinnt das antwortende Informatiksystem aus den Fragen? Welche Informationen können daraus abgeleitet werden?

Der Informationsbegriff erweist sich also ziemlich eindeutig im Bereich „Informatik und Gesellschaft“ als wirksam. Wir sollten ihn da auch belassen. Diese Beschränkung beschneidet m. E. nicht seine Bedeutung, im Gegenteil: Wenn ein Begriff die Ausrichtung eines Schulfachs deutlich akzentuieren kann, dann ist das nicht wenig. Es ist viel.

Wenn der Informationsbegriff in Bezug auf die Daten nicht sehr ergiebig, in Bezug auf den Bereich Informatik und Gesellschaft aber hilfreich ist, muss die Bedeutung des Inhaltsbereichs „Daten“ aus sich selbst heraus folgen – sonst wird es unter Allgemeinbildungsaspekten schwierig, die Existenz dieses Bereichs zu rechtfertigen. Bei den genannten Inhaltsbereichen dieses Gebiets finden sich neben dem etwas eingestreut wirkenden Informationsbegriff die klassischen Themen eines Standardbereichs der Fachinformatik: des Gebiets *Algorithmen und Datenstrukturen*. Mir scheint, dass die Fachinformatik hier ihre Inhalte aus gutem Grund anders strukturiert als die aktuelle Informatikdidaktik: offensichtlich sind die Berührungspunkte zwischen Daten und Informationen sehr übersichtlich, aber zwischen Algorithmen und Datenstrukturen kann es eigentlich keine Trennung geben, weil jeweils der andere Bereich für den einen unverzichtbar ist. Deutlich wird dieses auch in der GI-Forderung nach *„modellieren und implementieren als durchgängige Methode“*. Auch wenn die fachwissenschaftliche Strukturierung der Inhaltsbereiche für die Fachdidaktik keine zwingende Vorgabe ist, sollte man sie berücksichtigen, denn „unsinnig“ ist sie sicher nicht.

Die Frage stellt sich etwas anders: *Welche Teile der fachwissenschaftlichen Grundausbildung sind zu einem bestimmten Zeitpunkt für die allgemeinbildend orientierte Fachdidaktik relevant?* Oder noch anders: *Wenn „früher" bestimmte Fachfragen auch für Schulen bedeutsam waren, weil die technische und fachliche Entwicklung zu diesem Zeitpunkt erst einen bestimmten Stand erreicht hatte, dann ergibt sich daraus noch keine zwingende Begründung für die Relevanz dieser Fachthemen zu einem späteren Zeitpunkt.* Als Beispiel mögen die linearen Datenstrukturen (*Stapel, Schlange*, …) dienen: In der Fachausbildung sind sie nach wie vor relevant und dort eng mit den auf ihnen arbeitenden Algorithmen verknüpft. In der Schule hatten sie ihre Bedeutung, weil ohne ihre Implementierung anspruchsvolle selbstständige Schülerarbeit kaum zu realisieren war. Mit den heute zur Verfügung stehenden Werkzeugen muss allerdings gefragt werden, ob die Implementation dieser Strukturen noch notwendig ist. Stehen gut visualisierbare Listen zur Verfügung, dann unterscheiden sich die linearen Strukturen eigentlich nur durch den Ort ihres Zugriffs, dem Listenanfang oder -ende – und den kann man sehen. Es ist intuitiv klar, was welche Operation bewirkt.

Im Bereich der Schule ist es nun kaum sinnvoll, die Verarbeitung der Daten als Selbstzweck zu betrachten. Erforderlich ist wieder ein Kontext, aus dem sich der Bedarf nach deren Transformation ergibt. Daten erhalten dadurch einen Sinn, ihre Verarbeitung erfolgt zu einem bestimmten Zweck. Ohne diesen Kontext ist der für die Begründung des Schulfaches Informatik zentrale Kompetenzerwerb aus dem Bereich „Begründen und Bewerten" auch kaum realisierbar. Der Kontext ist also ernst zu nehmen. Er steht gleichrangig neben den Fachinhalten. Pseudokontexte, die nur dazu dienen, möglichst schnell zu den Fachinhalten zu kommen, sind eher kontraproduktiv. Wenn der Kontext offensichtlich bedeutungslos ist, dann wird diese „Unsinnigkeit" leicht auf die Fachinhalte übertragen, die den Lernenden folglich ebenfalls bedeutungslos erscheinen.

Aus dem Kontext stammen Daten, die in veränderter Form in den Kontext zurückfließen. Als Musterbeispiel dafür mag das *Physical Computing* dienen, bei dem Sensorwerte vom Informatiksystem erfasst und zur Erzeugung der Aktoren-Steuerungsdaten benutzt werden. (*„Wenn es regnet, dann werden die Fenster geschlossen." „Wenn der Zug kommt, ist die Schranke besser unten."*) Das Beispiel zeigt auch, dass einfache Zahlenwerte als Daten durchaus ihre Bedeutung haben können. Sie haben diese Bedeutung aber nicht „an sich", sondern gewinnen sie im gegebenen Rahmen. Das Beispiel zeigt auch, dass die Lernenden nicht unbedingt Aufgaben lösen müssen, die der Unterrichtende gestellt hat, sondern Probleme bearbeiten können, die sie selbst aus dem gegebenen Kontext abgeleitet haben. Sie bearbeiten dann nicht Übungsaufgaben, sondern betätigen sich als Problemlöser, hier als kleine Konstrukteure, die anderen Menschen das Leben leichter machen oder Katastrophen verhindern. Die Transformation der Daten ist nicht Selbstzweck, sondern Mittel auf dem Weg zu einem selbst gesteckten Ziel.

Der Kontext muss nicht unbedingt real (wie beim *Physical Computing*) oder simuliert (durch die Multimedia-Eigenschaften der visuellen Programmiersprachen) auftreten. Er kann auch eine Geschichte sein, aus der sich die informatische Fragestellung ergibt. Die Berechnung eines bestimmten Prozentsatzes nach einem vorgegebenen Verfahren muss nicht jeden motivieren. Stellt man aber die Frage nach dem Beitrag z. B. Deutschlands an den Schäden eines Hurricanes an einem ganz anderen Ort[10], dann erhält eine einzelne Zahl eine immense Bedeutung, selbst wenn wir sie nur ansatzweise bestimmen können. Auch die Erkennung einer Ziffer auf einem Bild wird für Lernende interessant, wenn sich z. B. das Problem der KFZ-Nummernschilderkennung aus einer spannenden Geschichte oder einem aktuellen Fall ergibt. Egal, wie der

[10] Friedericke Otto, World Weather Attribution, https://wwa.climatecentral.org/

Kontext gewählt wird: seine Bedeutung für die Motivation der Lernenden erfordert, dass die Entwicklungsumgebung ihn berücksichtigen kann, durch Grafiken, Klänge, Animationen. Damit diese Darstellung nicht die eigentlichen Fachinhalte verdrängt, muss die Kontextdarstellung sehr einfach zu bewerkstelligen sein.

Strukturierte gleichartige Daten treten in direkter Form meist als Zeichenketten oder Bilder auf. Deshalb gibt es dafür eigene Datentypen. Lineare Datenmengen treten entweder als Folgen der Ein-/Ausgabewerte *(Datenströme*) auf oder als Zeichenketten, die zu bestimmten Zwecken (*Kryptografie, …*) transformiert werden. Beide Möglichkeiten zeigen, dass sich die Einbettung in einen sinngebenden Kontext wie von selbst ergibt. Bei Bildern folgt meist aus der Problemstellung direkt die erforderliche Transformation. Als Beispiele mögen *Bildverbesserung, Farbänderungen, Kantenerkennung und daraus folgend Gegenstandserkennung, Klassifizierung von Bildern* usw. dienen. Da die Repräsentation dieser Datenmengen als listenartige Strukturen bzw. Tabellen intuitiv klar ist, stellt deren algorithmische Behandlung meist kein großes Problem dar. Etwas anders sieht es mit der Kontrolle der entwickelten Algorithmen aus. Da diese Strukturen viele Daten enthalten können, ist die leichte Visualisierbarkeit, aus der sich der momentane Zustand der Datenmenge ergibt, für Lernende entscheidend. In Schulen kommt es also nicht so sehr auf die (immer vorhandenen) algorithmischen Bausteine an, sondern auf die Darstellbarkeit ihrer Wirkungen.

Die direkte „Datenverarbeitung“ spielt in der Schule keine große Rolle mehr, weil spezielle Werkzeuge wie Datenbanksysteme die Teilaufgaben übernommen haben. Die Daten sind deshalb weitgehend Elemente von Modellen, in denen sie die Teile der Systeme beschreiben und Zusammenhänge darstellen. Zusammen mit der Forderung nach einem für die Lernenden sinnvoll erscheinenden Kontext ergibt sich daraus, dass der Themenbereich „Daten“ überwiegend in einen Themenbereich „Modellierung“ eingebettet sein sollte.

1.2 Informatik und Medienbildung

In Schulen und Universitäten wird im Rahmen der „Digitalisierungsoffensive“ heftig über die Vermittlung von Medienkompetenz diskutiert. Da der Begriff „Digitalisierung“ offensichtlich auch die Informatik betrifft, sollte diese sich an der Diskussion beteiligen. Lehranstalten müssen sich gut überlegen, was genau ihr Beitrag zu einer Gesamtbildung ist. Einerseits gewinnen Kinder und Jugendliche Kenntnisse und Erfahrungen auch – und in vielen Bereichen überwiegend – außerhalb dieser Institutionen, andererseits sollten die Ziele „Bildung“ und „Ausbildung“ scharf unterschieden werden. Jugendliche müssen nicht den Umgang mit aktuellen professionellen Werkzeugen beherrschen, das können sie getrost den Erwachsenen überlassen. Sie müssen aber darauf vorbereitet werden, deren Rolle bei künftigen Tools zu übernehmen.

Es wird und wurde oft argumentiert, dass die Lernenden den Umgang mit modernen Medien erlernen müssen, um die „Angst vor ihnen“ zu verlieren. Ich halte das für abwegig. Erstens haben Kinder und Jugendliche vor Medien normalerweise keine Angst, sondern sie sind neugierig darauf. Zweitens lernen sie den Umgang damit schnell und unkompliziert nebenbei von anderen und durch Gebrauch. Die Angst ist eher auf Seiten der Älteren, die nicht mit dieser Technik aufgewachsen sind und sich deshalb unsicher damit fühlen. Die derzeit Älteren sollten sich daran erinnern, dass in ihrer Jugend seitens der damals Älteren diskutiert wurde, wie ihnen der Umgang mit mausgesteuerten Oberflächen behutsam nahezubringen wäre, um ihnen die Angst davor zu nehmen. Wir können daraus lernen, dass der Umgang mit aktueller Technik, etwa Smartphones, nebenbei erlernt wird, dass dieses aber offensichtlich nicht automatisch dazu führt, die zukünftige Technologie ebenso unkompliziert zu nutzen.

> *Folgerung: Die Lernenden müssen befähigt werden, die Grundlagen zukünftiger Technologien zu verstehen und sich deren Nutzung zu erschließen. Dafür benötigen sie allgemeine Kenntnisse der fachlichen Grundlagen der Informationstechnologien, nicht aber Spezialwissen der aktuellen Technik.*

Es ist wohl selbstverständlich, dass Mediennutzung nicht mit Medienkonsum gleichzusetzen ist. Der passive Gebrauch von Medien welcher Art auch immer, also z. B. schlichtes „Glotzen“, kann nicht Ziel des Bildungssystems sein. Wenn wir uns mit Medien beschäftigen, dann müssen diese in einem Kontext auftreten, der Lernende aktiviert.

> *Folgerung: Die Lernenden müssen befähigt werden, Werkzeuge z. B. zur Erstellung von Medien problemabhängig auszuwählen und einzusetzen. Dafür müssen sie lernen, selbstständig Probleme zu lösen.*

Die Erziehung zum selbstständigen Problemlösen wird zumindest in Schulen meist nicht gerade als zentrale Aufgabe gesehen. Kreative Fächer wie Kunst, Musik und teilweise die Sprachen streben dieses wenigstens manchmal an. Meist steht allerdings braves Lernen im Vordergrund. Die Informatik stellt nun Werkzeuge bereit, mit deren Hilfe eigene Ideen schon in relativ rudimentärer Gestalt realisiert, getestet und verbessert werden können. Es wäre eine vergebene Chance, wenn das Fach nicht einen für die Lernenden kreativen Unterricht realisieren würde. Das wird allerdings nur funktionieren, wenn die Lehrenden selbst Erfahrungen im selbstständigen, kreativen Problemlösen haben, und wenn sie den Lernenden Entsprechendes zutrauen. Wenn die Lehrenden die informatischen Inhalte selbst nur „brav gelernt“ haben, dann wird es mit der Kreativität im Unterricht nicht klappen. Soll in Schulen das selbstständige Problemlösen angestrebt werden, dann sollte und muss das auch Konsequenzen für die Lehrendenausbildung an den Universitäten haben.

> *Folgerung: Die Lehrenden müssen befähigt werden, kreativen Unterricht zu planen und zu realisieren. Dazu muss in ihrer eigenen Ausbildung Gelegenheit und Platz geschaffen werden.*

Moderne Medien wie z. B. die sozialen Netzwerke haben das soziale Leben, die Kommunikation usw. teilweise tiefgreifend verändert. Die Konsequenzen sind kaum abzusehen, während dieser Prozess noch läuft. Sehr viel weniger waren sie abzusehen, bevor er gestartet wurde. Ich hielte es deshalb für eine komplette

Überforderung der Lehrenden, zu verlangen, dass diese die tatsächlichen gesellschaftlichen Folgen von Informatiksystemen, zu denen die Auswirkungen der digitalen Medien ja zählen, im Unterricht behandeln. Das wäre auch nicht zielführend, weil der Blick auf die eingetretenen Folgen notwendig rückwärtsgewandt ist. Was man aber sehr wohl verlangen kann, ist, zu zeigen, dass der Gebrauch von Informatiksystemen gesellschaftliche Folgen hat und dass diese sehr davon abhängen, wie die Systeme gestaltet werden. Unterschiedliche Problemlösungen haben also unterschiedliche Folgen – und umgekehrt: Sind bestimmte Folgen unerwünscht, dann wird es meist möglich sein, eine andere technische Problemlösung zu finden.

> *Folgerung: Die Lernenden müssen erfahren, dass es zu gestellten Problemen fast immer unterschiedliche Lösungen gibt. Sie sollten sich Gedanken über deren Auswirkungen machen, die natürlich nicht abschließend sind. Sie lernen dabei, dass diese Auswirkungen nicht gegeben, sondern gestaltbar sind.*

Was hat das mit *Snap!* zu tun?

Grafische Programmierumgebungen wie *Snap!* beinhalten nicht nur die algorithmischen Komponenten, sondern sind in eine mediale Umgebung eingebettet, die die Nutzung von Grafik, Sound, … nicht nur gestattet, sondern erfordert. Wird ein Problem bearbeitet, dann können und sollten Kameras und Grafikprogramme benutzt werden, um die entsprechenden Kostüme zu erzeugen und Kostümwechsel zu ermöglichen, die den aktuellen Zustand des Systems visualisieren. Soundprogramme gestatten es, den Verlauf selbst zu kommentieren, Musik zu bearbeiten und einzufügen oder selbst zu gestalten. Und natürlich müssen die Ergebnisse präsentiert werden, weil der Produktstolz ein wichtiges Motiv für die engagierte Arbeit ist und das Interesse an den Ergebnissen der anderen groß. *Snap!* unterstützt gerade Präsentationsaspekt durch die neue Möglichkeit, zwischen mehreren Bühnen zu wechseln.

Snap! gestattet algorithmisches Problemlösen auf hohem Niveau, aber es ermöglicht nicht nur den analytischen Zugang, sondern auch den spielerischen, den experimentellen, den kreativen, … Was es nicht gestattet, ist Passivität, denn von allein passiert nichts. Medien sind wesentliche Systembestandteile z. B. zur Visualisierung der Ergebnisse – und sie können auch selbst das Ergebnis sein. *Snap!* bietet deshalb die Chance, modellhaft Problemlösungen für aktuelle Probleme zu konstruieren, z. B. auch und gerade im medialen Bereich. Durch das selbst erstellte algorithmische Gerüst des Modells entsteht Verständnis für die beobachteten Abläufe im realen Vorbild. Die Erfahrung, diese Erkenntnisse selbst gewinnen zu können, ermöglicht die aktive, kritische Auseinandersetzung mit zukünftiger Technologie. Die Beispiele des vorliegenden Buches sollen zeigen, dass dieses auf vielen Gebieten mithilfe elementarer Methoden möglich ist. Sie sollen ermutigen, selbst loszulegen. 😉

1.3 Objekte und Vererbung durch Delegation

Werden etwas umfangreichere Probleme bearbeitet, dann wächst auch die Zahl der zu lösenden Teilprobleme. Oft lassen sich diese zu Gruppen zusammenfassen, die konkreten *Objekten* zuzuordnen sind. Ein wichtiger Aspekt dieser Arbeitsweise ist, dass sich so arbeitsteilige Teamarbeit gut realisieren lässt, bei der die unterschiedlichen Teams Objekte erzeugen, die Teilaufgaben lösen. Die objektorientierte Arbeitsweise wird oft realisiert, indem *Klassen* erzeugt werden, die das Verhalten einer Gruppe ähnlicher Objekte beschreiben. Von diesen Klassen werden dann *Instanzen* (Exemplare) erzeugt, die die Probleme lösen sollen. Das Vorgehen ist weitgehend top-down und erfordert einiges an Abstraktion. Für Anfänger geeigneter ist das in *Snap!* verwandte *Prototypen*-basierte Vorgehen, bei dem für jede Objektgruppe ein Beispiel, der Prototyp, erzeugt wird, der schrittweise entwickelt und getestet wird. Ist man mit dem Ergebnis zufrieden, dann werden weitere Objekte dieser Art durch Vervielfältigung (*Klonen*) des Prototyps abgeleitet.

Zur Objektorientierten Programmierung gehört zentral das Konzept der Vererbung, das durch Klassen oder per Delegation realisiert werden kann. Im Originalartikel von Lieberman[11], der das Prototypen-orientierte Vorgehen bei der Delegation schon sehr früh beschreibt, 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. In *Snap!* arbeiten wir überwiegend nach diesem Prinzip, das weiter unten detailliert vorgestellt wird.[12]

In *Snap!* werden Prototypen als Sprites erzeugt und mit den gewünschten Attributen und Methoden ausgestattet. Ist deren Verhalten genügend erprobt worden, dann können Klone 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. Stoppt das Programm, dann werden alle dynamisch erzeugten Klone gelöscht, was segensreich ist.

Ein Klon erbt anfangs (fast) alle lokalen 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 Prototyps. Löscht man die Überschreibungen wieder, dann erscheinen die geerbten.

[11] Lieberman, Henry: Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems, 1986, http://web.media.mit.edu/~lieber/Lieberary/OOP/Delegation/Delegation.html
[12] Will man es unbedingt, dann kann auch ein Klassensystem implementiert werden.

2 Zu *Snap!*

2.1 Was ist *Snap!*?

Snap![13] wurde (und wird) von *Brian Harvey* und *Jens Mönig* für das Projekt *Beauty and Joy of Computing*[14] entwickelt und im Internet frei zur Verfügung gestellt. Da das System im Browser läuft, benötigt es keinerlei Installation und funktioniert auf fast allen Geräten[15]. Es ähnelt von der Oberfläche und dem Verhalten her *Scratch*[16], einer ebenfalls freien Programmierumgebung für Kinder, die am *MIT*[17] 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 bei der Ausbildung von Informatik-Studierenden eingesetzt wird. Eingeführt werden sie z. B. in einem berühmten Lehrbuch von Harold Abelson sowie Gerald und Julie Sussman[18]. *Snap!* ist damit eine voll entwickelte Programmiersprache, die folgerichtig auch in (fast) allen Problembereichen eingesetzt werden kann. Für die meisten ist sie inzwischen auch ausreichend schnell. Das ist nicht selbstverständlich und war ein Manko ihrer Vorgänger. Grafische Sprachen sind weitgehend damit beschäftigt, den Systemzustand zu kontrollieren und es so z. B. zu gestatten, Endlosschleifen zu unterbrechen oder Zugriffsfehler auf Datenstrukturen zu „tolerieren". Für die eigentliche Programmausführung bleibt dann wenig Zeit.

Snap! 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" Programme weitgehend verhindert. *Snap!* 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.

Snap! ist außerordentlich „friedlich": Fehler führen nicht zu Programmabstürzen, sondern werden durch das Auftauchen einer roten Markierung um die Kacheln angezeigt, die den Fehler verursachten – ohne dramatische Folgen. Die benutzten Kacheln, zu denen auch die neu 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.

Snap! ist anschaulich: sowohl die Programmabläufe wie die Belegungen der Variablen lassen sich bei Bedarf am Bildschirm anzeigen und verfolgen. Damit ist es z. B. hervorragend für Simulationen geeignet.

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

---

[13] https://snap.berkeley.edu/snap/snap.html
[14] https://bjc.berkeley.edu/
[15] Gemeint sind natürlich Computer, Tablets, Smartphones, …
[16] http://scratch.mit.edu/
[17] Massachusetts Institute of Technology, Boston
[18] Abelson, Sussman: Struktur und Interpretation von Computerprogrammen, Springer 2001

Snap! 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.

Snap! ist erstklassig: alle benutzten Strukturen sind *first-class*, lassen sich also Variablen zuweisen oder als Parameter in Blöcken verwenden, können das Ergebnis eines Funktionsblocks 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 beinhaltet das Logo von *Snap!* dasselbe stolze Lambda, das sich früher bei *Alonzo*, dem Maskottchen von *BYOB*, als Tolle im Haarschopf fand.

Alonzo

2.2 Was ist *Snap!* nicht?

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

Snap! dient in erster Linie zur Arbeit auf dem Gebiet der Algorithmen und Datenstrukturen, aber auch in der Browserumgebung lassen sich wesentliche Bereiche der Informatik wie der Zugriff auf Dateien oder Hardware einbetten, manchmal über Bibliotheken. Das Mikrophon und die Kamera des Computers werden direkt angesprochen, und der eingebaute *http*-Block gestattet ziemlich einfach Zugriffe auf das Internet und damit z. B. über zwischengeschaltete Server die Nutzung von Datenbanken oder externer Hardware.

Da der Code von *Snap!* frei zur Verfügung steht, gibt es unterschiedliche Modifikationen. Ob das Fluch oder Segen ist, wird sich zeigen.

2.3 Der *Snap!*-Bildschirm

Der *Snap!* –Bildschirm besteht unterhalb der Menüleiste aus sechs Bereichen[19].

- 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 des Knopfs 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 *Snap! Sprite* genannt – sowie einige seiner Eigenschaften angezeigt. Den voreingestellten Namen des Sprites kann – und sollte – man hier ändern.
- Darunter befindet sich ein Bereich, in dem sich je nach Reiterkarte die *Skripte*, *Kostüme* und *Klänge* des Sprites bearbeiten oder erzeugen lassen.
- Rechts-oben befindet sich das Ausgabefenster, in dem sich die Sprites bewegen: die *Bühne*. Diese kann mithilfe der darüber befindlichen Buttons, dem Eintrag im Werkzeugmenü (*Stage size …*), einem entsprechenden Befehlsblock oder durch schlichtes „Ziehen“ mit der Maus in ihrer Größe verändert werden. Setzt man das Auswahlhäkchen vor den Variablennamen in der *Variables*-Palette, dann werden die Variablen auf der Bühne angezeigt, ggf. mit einem *Slider*, der das einfache Verändern der Werte ermöglicht. Da Variable alles enthalten können (Zahlen, Texte, Listen, Sprites, Programme, …), kann der Zustand dieser Größen jederzeit visualisiert 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. Links neben den Sprites wird ein Symbol der Bühne, oder, falls vorhanden, die Symbole mehrerer Bühnen angezeigt.

[19] Die Aufteilung der Bereiche kann mit den  verändert werden.

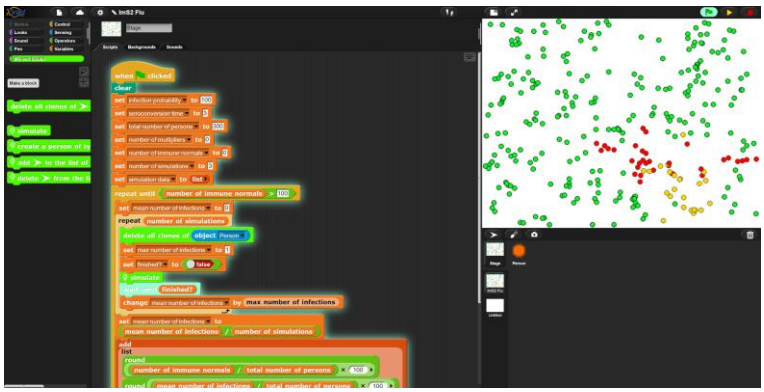
Auch zwischen diesen kann durch Anklicken gewechselt werden. Zu jeder Bühne gehört ein eigenes Projekt, das von denen der anderen Bühnen unabhängig ist. Allerdings kann man Daten zwischen den Projekten austauschen.

- 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. Eine Möglichkeit besteht darin, die Sprache einzustellen. Ich empfehle trotzdem, bei der englischen Version zu bleiben, da so eigene, in Deutsch bezeichnete Blöcke, von den systemeigenen auf den ersten Blick zu unterscheiden sind.
- 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.

2.4 Beispiel für Umsteiger: Grippe

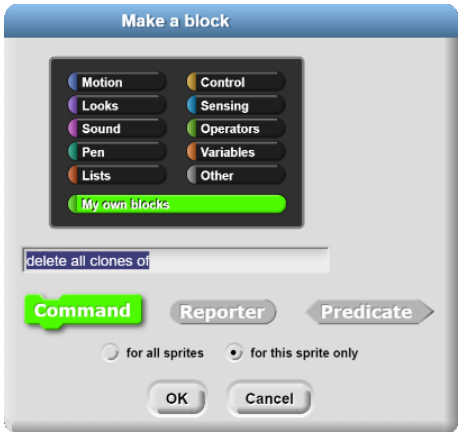
Altersstufe: *Sekundarstufe II* Material: *Flu*

Das Beispiel simuliert die Ausbreitung einer Grippeepidemie unter unterschiedlichen Bedingungen. Es dient zu einer schnellen Übersicht über wesentliche Möglichkeiten von *Snap!* und ist insbesondere für erfahrene Programmierer/innen gedacht. Einsteiger sollten lieber zuerst die nächsten Kapitel lesen.



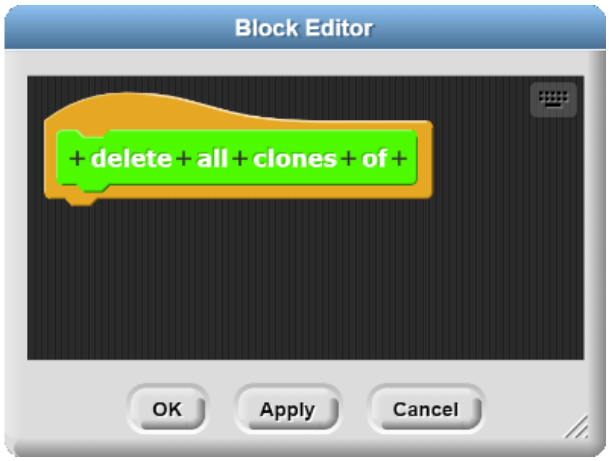
Es wird gefragt, welcher Anteil und welche besonderen Personengruppen in einer Population zu impfen wären, wenn die Ausbreitung einer Grippeepidemie gestoppt werden soll. Die Frage ist gar nicht so einfach zu beantworten, denn das Ergebnis hängt von verschiedenen Parametern ab: die *Infektionswahrscheinlichkeit* gibt an, wie wahrscheinlich die Infektion einer gesunden Person bei einem Kontakt mit einer kranken ist, die *Serokonversionszeit* ist die Zeit zwischen Infektion und Immunisierung, die *Anzahlen* von gesunden und erkrankten Personen zu Beginn der Simulation bestimmt die Anzahl der Kontakte zwischen diesen, und die Zahl der *Multiplikatoren* gibt an, wie viele Personen in der Population besonders viele Kontakte bzw. mit besonders weit auseinanderliegenden Gruppen Kontakt haben. Infiziert sich einer von ihnen, dann wird die Krankheit z. B. schnell in weit entfernte Gebiete getragen. Da Kontakte, Infektionen usw. zufallsgesteuert sind, werden wir nur dann tragfähige Ergebnisse erzielen, wenn wir die Simulation jeweils mit gleichen Parameterwerten mehrfach durchführen – und dann bleibt immer noch zu diskutieren, welche Werte überhaupt „Ergebnisse“ im genannten Sinn darstellen. Das Thema eignet sich deshalb hervorragend für ein kleines Unterrichtsprojekt. Eine „Steuergruppe“ entwickelt die übergeordneten Skripte, die wir hier der Bühne (*Stage*) zuordnen wollen. Sie stimmt mit den anderen Gruppen die Aufgabenverteilung ab. Die anderen Gruppen entwickeln Hilfsmethoden sowie die Prototypen *Person* und *Graph* mit jeweils eigenen Bühnen, die fast unabhängig voneinander sind, und machen sich Gedanken über den Datenaustausch.

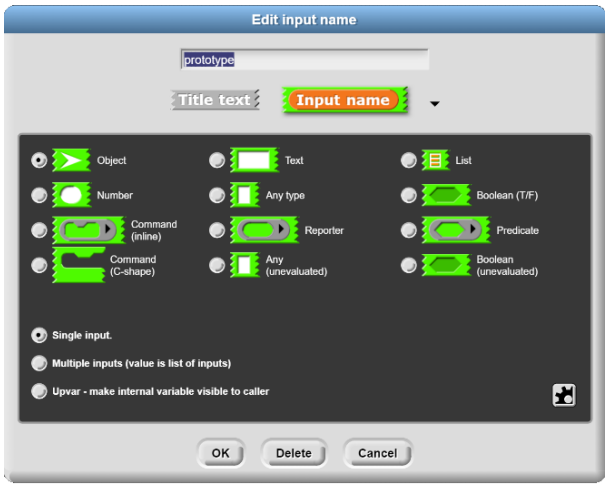
2.4.1 Eigene Methoden schreiben



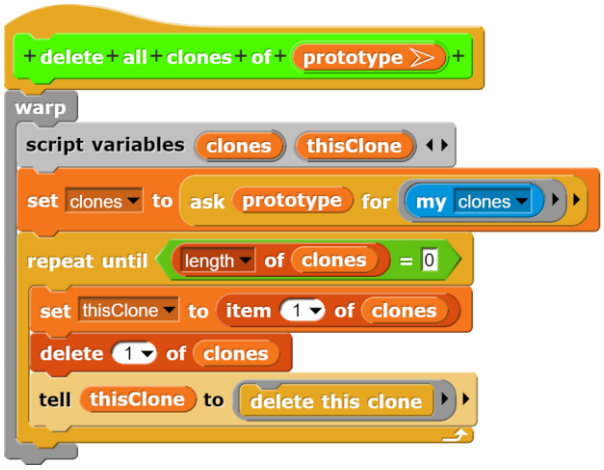
Es ist oft notwendig, die erzeugten Klone eines Prototyps wieder loszuwerden, ohne das Programm zu beenden. Wir erreichen das hier durch eine neue lokale Methode *delete all clones of <a prototype>* der Stage. Es handelt sich um einen *Command*-Block, also einen Befehl, der (hier) einen Parameter hat. (Funktionsblöcke werden in *Snap! Reporter* genannt.) Neue Blöcke werden im Block-Editor geschrieben, der mit dem Button *Make a block*, den wir in den Paletten oder durch einen Rechtsklick auf die Skriptebene und dort im Kontextmenü finden, aufgerufen wird. Zuerst geben wir den Methodennamen an, wenn gewünscht mit Leer- und Sonderzeichen, wählen den Typ (*Command*, *Reporter* oder *Predicate*) und geben an, ob es sich um eine *globale* (*for all sprites*) oder *lokale* (*for this sprite only*) Methode handelt. Wir können auch die Palette wählen, in die der Block aufgenommen werden soll und deren Farbe er bekommt.

In diesem Fall erzeugen wir dafür vorher eine neue Palette (*category*) mithilfe des Dateimenüs (*New category…*), nennen sie *My own blocks* und wählen als Farbe ein optimistisches Grün, das die eigenen Blöcke deutlich von den vorgegebenen unterscheidet. Nach Drücken der Return-Taste öffnet sich der Block-Editor und der Blockname erscheint – mit +-Zeichen in den Zwischenräumen und Rändern. Dort können wir durch Mausklicks ein weiteres Menü öffnen, das es gestattet, Parameter (oder weitere Texte/Symbole) an diesen Stellen einzufügen und bei Bedarf zu typisieren. In unserem Fall klicken wir ganz rechts, geben den Parameterbezeichner *prototype* an und klicken zur Typisierung auf den kleinen Rechtspfeil. Danach öffnet sich eine Auswahlbox[20]. Wir wählen als Typ *Object* (den Pfeil), kommen wieder in den Block-Editor und ziehen die benötigten Befehle in dessen Skriptbereich.





Unsere Methode benutzt zwei Skript-Variable (*clones* und *thisClone*), die nur in diesem Block bekannt sind. Sie befragt den Parameter *prototype*, der später mit einer Referenz auf die „Urperson“ übergeben wird, nach seinen Nachkommen – das sind dann alle vorkommenden dynamisch erzeugten „Personen“[21]. Solange von diesen noch welche vorhanden sind, merkt sie sich die erste in einer der Skriptvariablen, löscht sie aus der Liste und bittet dann diese Person, sich selbst zu löschen, mit *tell <thisClone> to <delete this clone>*[22].

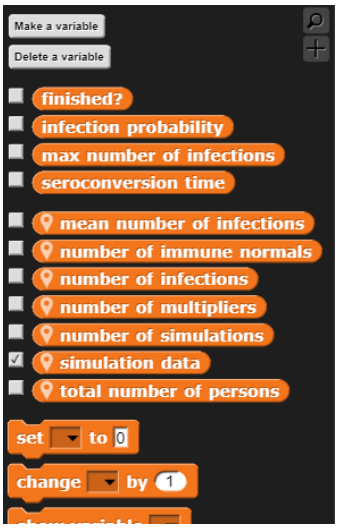


Aufgerufen wird die Methode, indem ihr ein Objekt (hier: Person) übergeben wird.

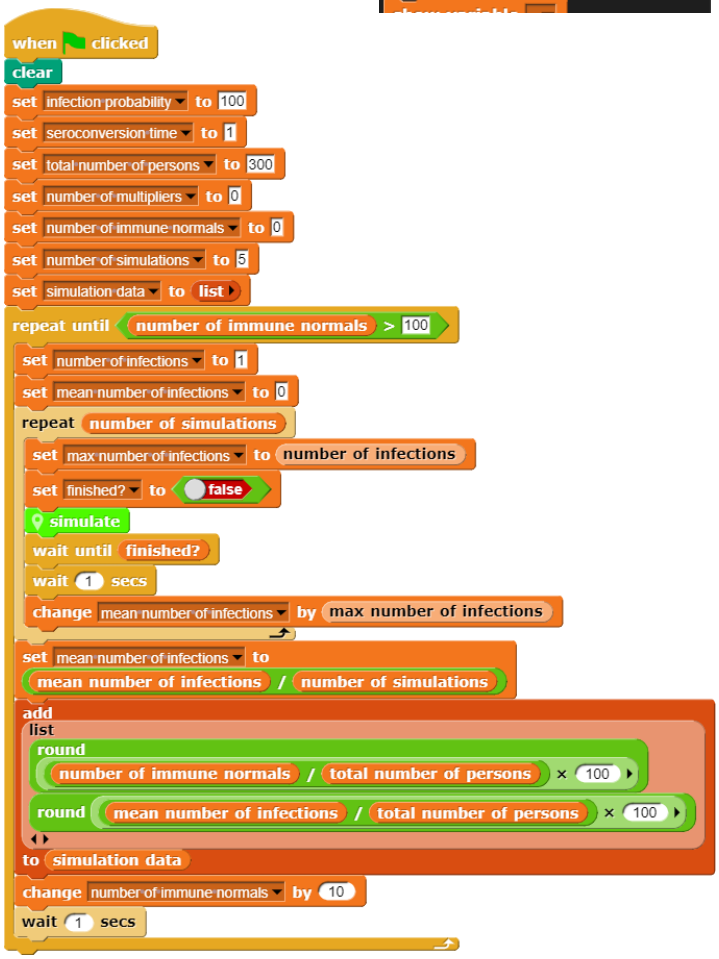
[20] Diese Box wird sehr ausführlich im Snap!-Reference-Manual beschrieben, das man erhält, wenn man oben-links im Fenster auf das Snap!-Symbol klickt.
[21] Die statisch über das Kontextmenü im Sprite-Bereich erzeugten Klone finden sich dort nicht.
[22] Der delete-Block findet sich nur in der Paletten der Sprites. Man erreicht ihn aber in der Stage über die Suchfunktion oben im Palettenbereich.

2.4.2 Elementare Algorithmik und Variable

Zur Festlegung der Parameter und anderer Steuerungswerte benutzen wir die *Stage*, die wir dazu im Sprite-Bereich anklicken. Diese reagiert auf die Botschaft *„Grüne Flagge angeklickt“*, indem sie die Anfangsparameter setzt und festlegt, welche Größen bei den Simulationen gemessen werden sollen. Danach werden entsprechende Simulationsläufe gestartet.



Im Detail: Eine Referenz auf den Prototyp *Person* können wir uns mithilfe des Blocks *object <Person>* aus der *Sensing*-Palette „angeln“. Bei Bedarf können wir sie, wie jede andere Größe in *Snap!*, in einer Variablen[23] speichern, die entweder global (*for all sprites*) oder lokal (*for this sprite only*) sein kann. Variable werden in der *Variables*-Palette über den Button *Make a variable* erzeugt. Dabei können wir auch gleich alle weiteren benötigten Variablen erzeugen, wobei die nur innerhalb der Stage benötigten als lokal gekennzeichneten werden. Man erkennt sie am „Marker“ vor dem Namen. Die anderen sind global. Globale Variable werden oben in der *Variables*-Palette angezeigt, danach folgen die aktuellen lokalen. Anschließend wird der Ausgabebereich gelöscht, einige Variable erhalten geeignete Anfangswerte und eine Liste namens *simulation data*, die die Simulationsergebnisse aufnehmen soll, wird gelöscht (*set <simulation data> to <list>*). Diesen Teil hätte man gut in einen eigenen Block auslagern können, aber da wir mit den Variablenwerten experimentieren wollen, ist es besser, wenn diese „auf dem Tisch“ liegen.

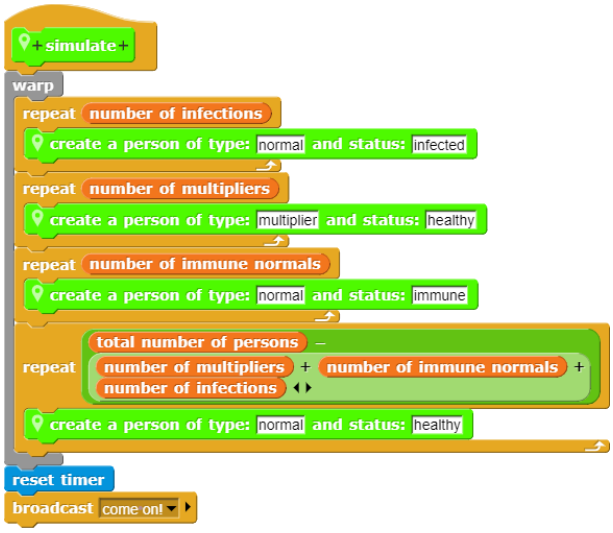


Im Folgenden wird die Zahl der anfangs Geimpften (die *number of immune normals*) von Null ausgehend bis auf 100 schrittweise erhöht. Die Kontrollstrukturen dafür finden wir in der *Control*-Palette. Für jeden Wert wird eine Reihe von Simulationsläufen durchgeführt und der Mittelwert aus den Ergebnissen (hier: die Maximalzahl von Infizierten) bestimmt. Die Variable *number of simulations* legt fest, wie oft das geschieht. Nach jedem Durchlauf werden die Ergebnisse prozentual in die *simulation data*-Liste eingetragen. Zum Schluss wird gebeten, eine Grafik aus diesen Daten zu erzeugen. Eine andere Arbeitsgruppe kann sich darum kümmern.

[23] Variablennamen unterliegen keinerlei Beschränkungen. Sie können also auch Leerzeichen, Sonderzeichen usw. enthalten.

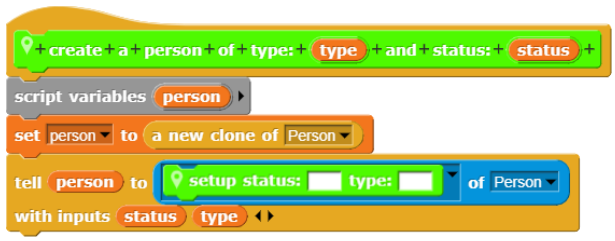
2.4.3 Objekte erzeugen

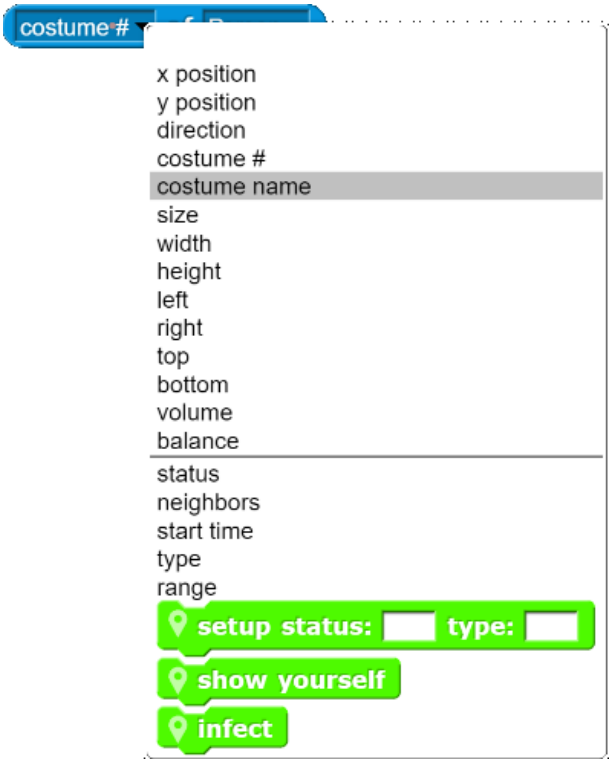
Im Steuerprogramm wird eine Methode *simulate* benutzt. In dieser werden einige Anfangswerte neu gesetzt und die entsprechende Anzahl von Personen erzeugt, die sich in Typ (*normal*, *multiplier*) und Status (*healthy*, *infected*, *immune*) unterscheiden. Um das Tempo zu erhöhen, geschieht das in einem *warp*-Block. Danach wird der Simulationslauf durch Senden der Nachricht *„come on!“*, die von allen Objekten im System „gehört“ wird, gestartet.

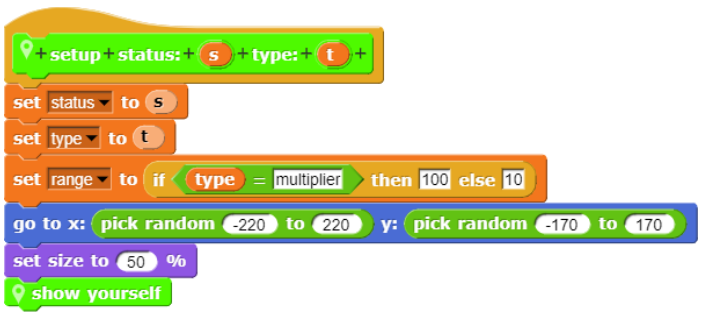


Wie erzeugt man Objekte?

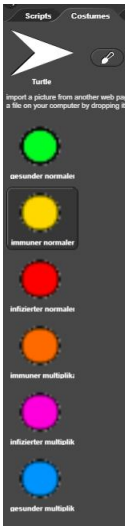
Wir vereinbaren in der dafür geschriebenen Methode *create a person of type <type> and status <status>* zuerst eine lokale Skriptvariable, der wir eine Referenz auf einen neu geschaffenen Klon des angegebenen Prototyps zuweisen. Danach ist der Klon vorhanden, sichtbar und unter dem Namen *person* erreichbar – ganz einfach. Die Klone sollen sich aber in Typ und Status unterscheiden. Dafür enthalten sie (hier) eine vom Prototyp geerbte lokale Methode *setup status: <status> type: <type>*. Diese müssen wir mit den übergebenen Parameterwerten aufrufen. Wir „sagen“ (*tell …*) deshalb dem Objekt *person*, dass es diese Methode ausführen soll. Da diese lokal für Personen ist, nehmen wir den *<attribute> of <object>* - Block aus dem *Sensing*-Menü, wählen im rechten Feld den Prototyp (hier: *Person*) aus und danach im linken Feld die gewünschte Methode (hier: *setup …*). Weil zwei Parameter anzugeben sind, erweitern wir den Block mit den kleinen Pfeiltasten und geben hinter *with inputs* den Status und Typ an. Der Block ist zu verstehen als *„person, führe bitte in deinem Kontext von Methoden und Variablen die übergebene Methode mit den angegebenen Parametern aus“*. Der Block ist äquivalent zur bekannten Punktnotation der OOP-Sprachen: z. B. `person.setup(status,type);`





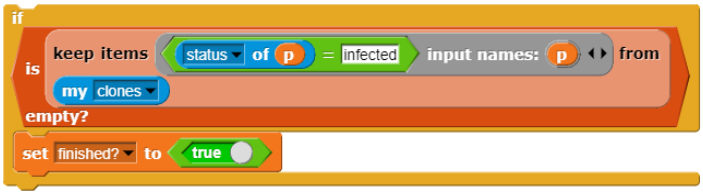


2.4.4 Mit Objekten kommunizieren



Wir kommen jetzt zu den eigentlichen Akteuren unseres Grippe-Projekts: den Personen. Diese werden durch kleine Kreise symbolisiert, deren Farbe ihren Status ausdrückt. „Normale“ Personen wuseln relativ kleinschrittig in ihrer Umgebung herum und treffen dabei auf die Nachbarn, an denen sie sich oder die sie anstecken können. Nach einer gewissen Zeit, der Serokonversionszeit, werden sie immun und stecken nicht mehr an, werden auch nicht mehr angesteckt. Geimpfte Personen sind von Anfang an immun. Einige der Personen sind „Multiplikatoren“, d. h. sie springen recht wild in der Gegend herum und können die Infektion schnell verbreiten. Sie sind ähnlich wie die Normalen, aber etwas anders farblich kodiert. Entsprechende Kostüme stellen wir im Grafikeditor oder einem Zeichenprogramm her und importieren sie in den *Costumes*-Bereich.

Nach der Erzeugung der Personen erhalten diese alle die Nachricht „*come on!*“, auf die sie reagieren, weil sie über einen *Hat*-Block aus der *Control*-Palette verfügen, der auf „come on*!*“ reagiert. Danach geraten sie in eine Schleife, die abbricht, wenn die globale Variable *finished?* den Wert *true* erhält. Das ist der Fall, wenn keine Infizierten mehr vorhanden sind, wenn die Liste der Klone, die noch infiziert sind, also leer ist.



In dieser Schleife werden die folgenden Aktionen wiederholt ausgeführt:

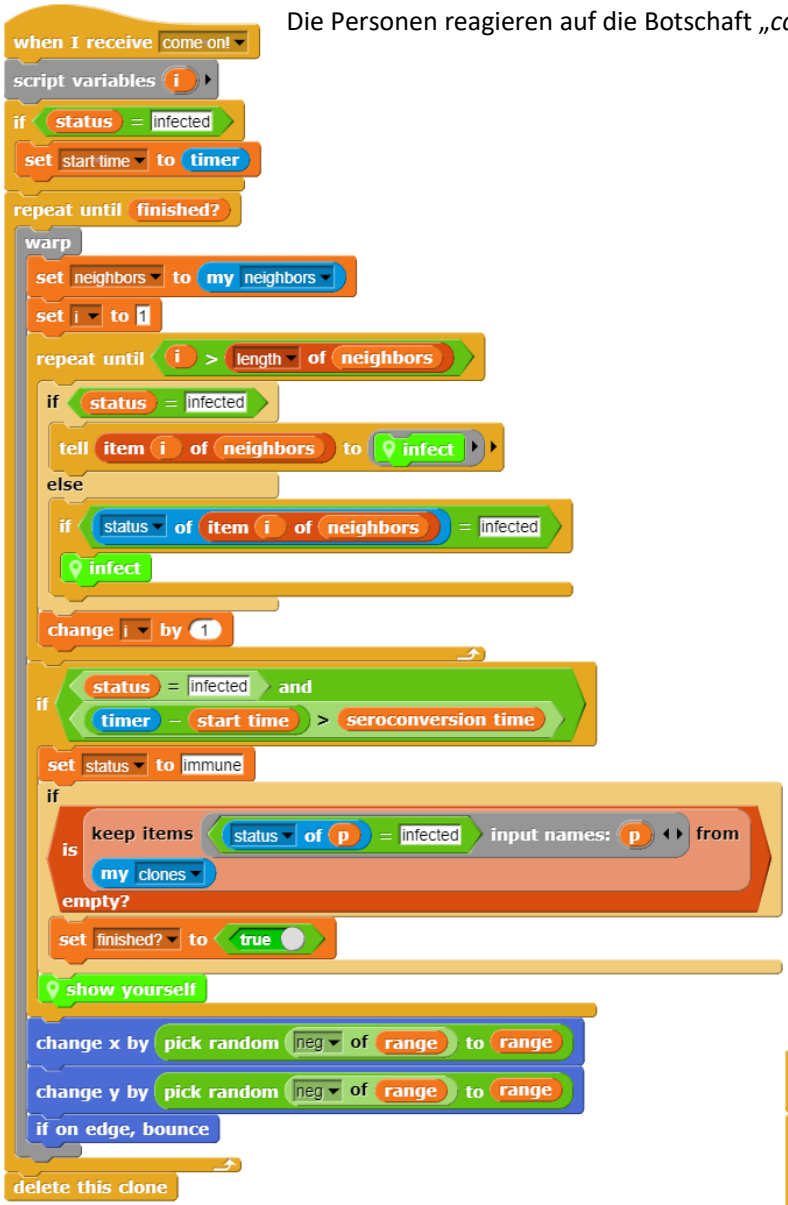
1. Es werden Objekte in der Nähe der Person gesucht und in der Liste *neighbors* gespeichert.
2. Alle verbleibenden Nachbarn werden ggf. infiziert oder infizieren die Person, wenn sie krank sind.
3. Es wird überprüft, ob die Person nach Ablauf der Serokonversionszeit immun werden muss. Die entsprechenden Variablenwerte werden geändert.
4. Danach bewegt sich die Person entsprechend ihres Typs.
5. Nach Beendigung der Schleife löscht sich der Klon selbst.

Da bei diesen Prozessen Daten zwischen den Personen ausgetauscht werden müssen und Methodenaufrufe der anderen Personen initiiert werden, zeigt das Beispiel einige Verfahren dafür:

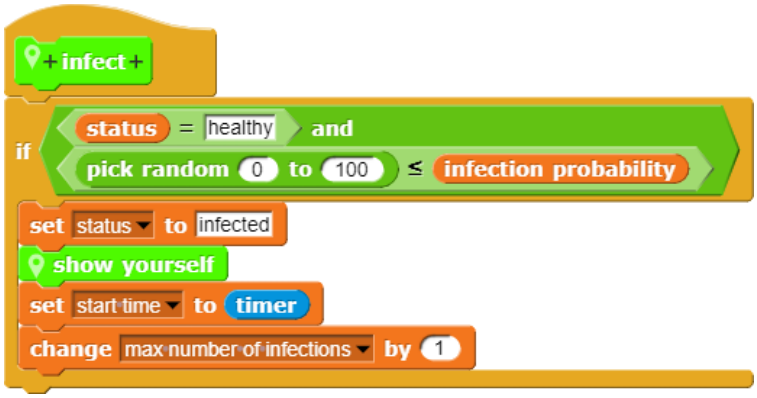
Der *tell <object> to <run this script>*-Block wird benutzt, um eine Person zu bitten, sich zu infizieren. Ruft man eine Funktion (die ein Ergebnis liefert) eines anderen Objekts auf, dann benutzt man den *ask <object> for <reporter>*-Block. Attribute und lokale Methoden anderer Objekte erhält man über den *my <attribut>*-Block aus der *Sensing*-Palette, den Sie schon kennengelernt haben. Hier fragen wir den Status eines Objekts ab, indem der *<attribute> of <object>*-Block im Kontext des anderen Objekts ausgeführt wird. Die Blöcke sind von einem grauen Ring umgeben (*ringified*), der anzeigt, dass der unevaluierte Code des Blocks übergeben wird und nicht sein aktuelles Ergebnis.

An zwei Stellen darunter werden lokale Methoden – grün dargestellt – im Kontext des Objekts ausgeführt. Das geschieht „ganz normal“ beim Erreichen des Blocks.

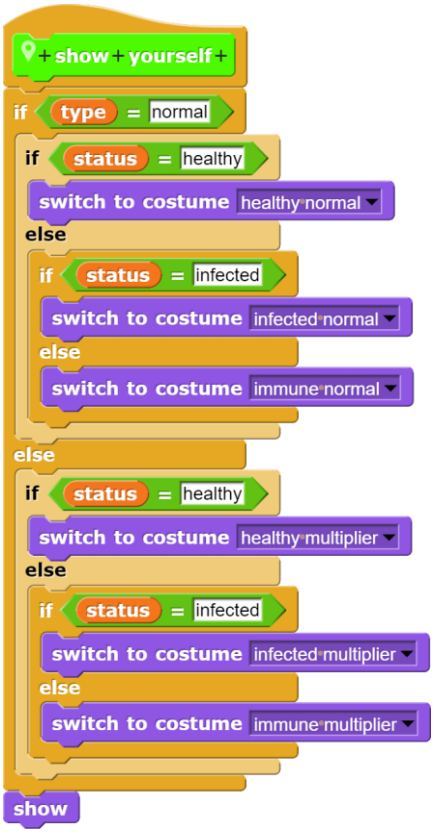
Die Personen reagieren auf die Botschaft „*come on!*“:



Die Methode *infect* infiziert ggf. das aktuelle Objekt und trägt frisch Infizierte in die entsprechende Liste ein. Danach wird das Aussehen des Objekts verändert.



Die Methode *show yourself* wählt das angemessene Kostüm aus.



2.4.5 Ein Diagramm zeichnen

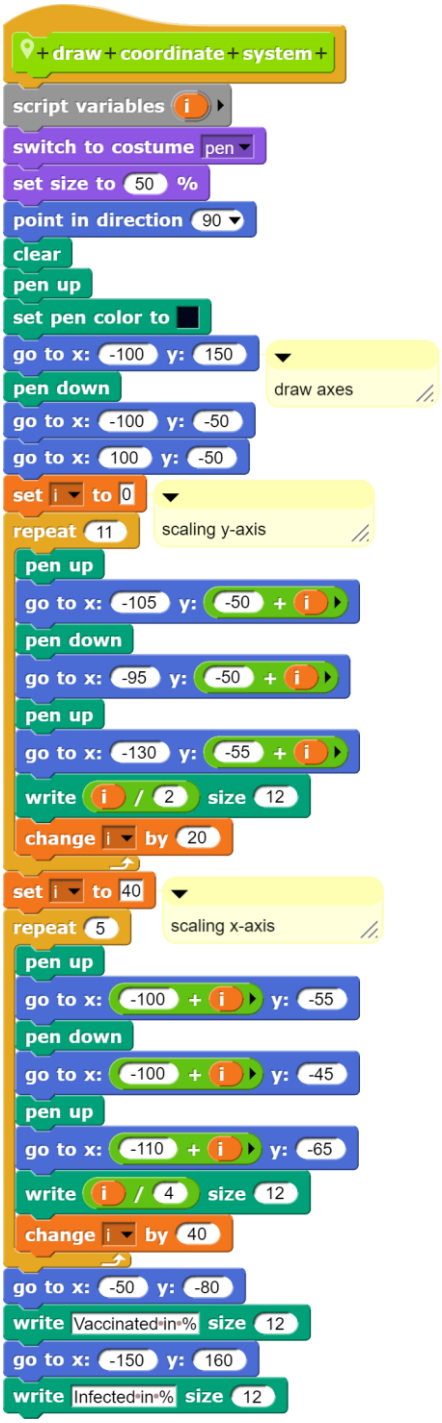
Zuletzt wollen wir unsere Ergebnisse in einem Diagramm darstellen lassen. Gemessen wurde die Anfangszahl der Geimpften (in %) und die Maximalzahl der Infizierten (in %). Wir erzeugen zu diesem Zweck eine zweite Bühne namens *Graph* mit *New scene* aus dem Datei-Menü[24]. Auf dieser existiert ein neues, zweites Projekt, das mit dem ersten nichts zu tun hat. Dessen Objekte, Variablen und Methoden sind in der zweiten Szene unbekannt. Wir können aber mit den *export-/import*-Funktionalitäten Objekte und/oder Skripte von einem Projekt zum anderen senden – also über Dateien. Zusätzlich können wir zwischen den Szenen wechseln und dabei auch Daten von einem Projekt zum anderen senden. Diesen *„internen“* Weg wollen wir gehen.

Wir erzeugen in der zweiten Szene ein Objekt namens *Pen*, dem wir als Kostüm einen schönen Stift spendieren. Diesen lassen wir zuerst ein Koordinatensystem auf den Bildschirm malen und beschriften. Die Blöcke dafür finden wir in der *Pen*-Palette.

Die ermittelten Daten liegen in Listenform als Variable *simulation data* vor. Sie werden nach Abschluss der Simulationen an die Szene *Graph* versandt. Diese Daten holt sich das Objekt *Pen* aus der Variablen *message* und speichert sie als *data*.

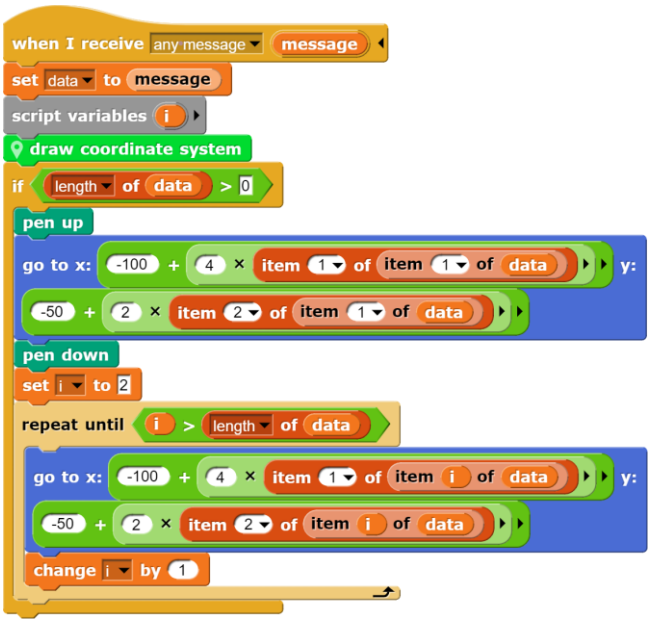
Stage simulation data

11	A	2
1	0	97
2	3	93
3	7	89
4	10	84
5	13	82
6	17	81
7	20	75
8	23	74
9	27	70
10	30	65
11	33	50

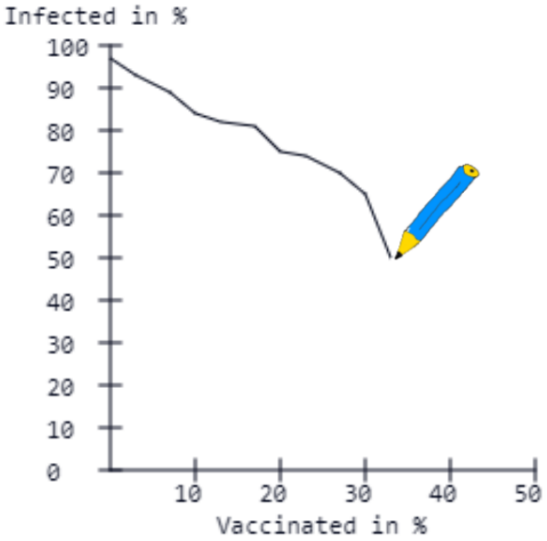


[24] nur, um das auch einmal zu zeigen 😉

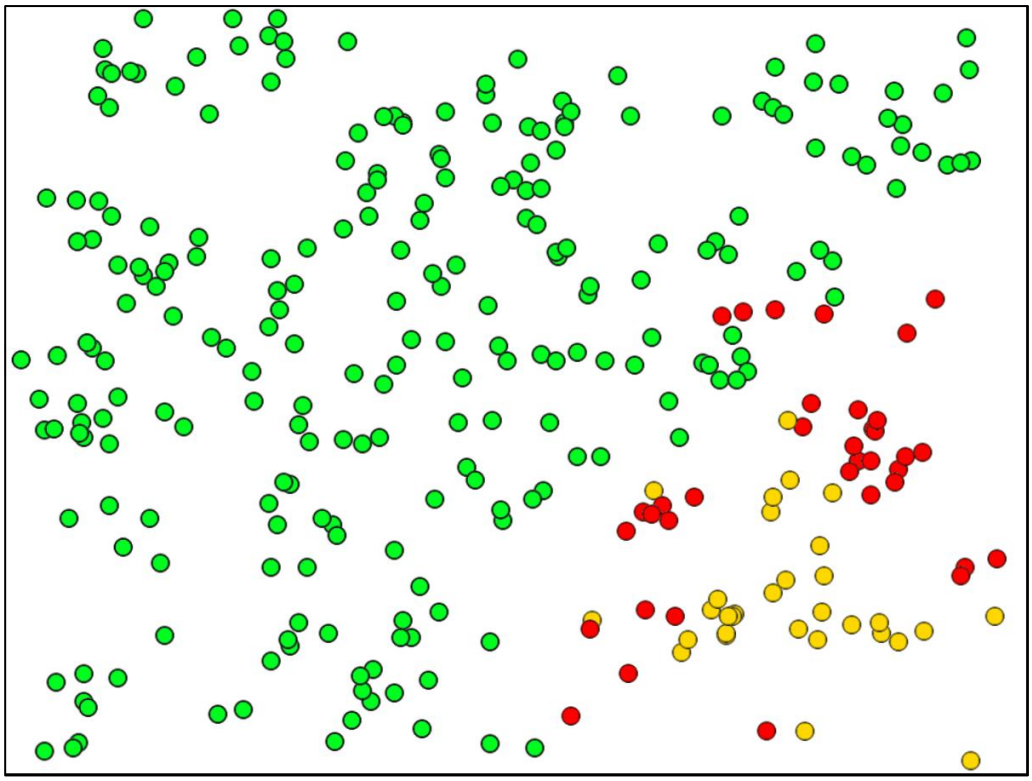
Danach werden die Datenpunkte in das Diagramm übertragen. Wir schicken den Stift zum ersten Datenpunkt, der durch eine Liste mit den zwei genannten Einträgen gegeben ist. Danach führen wir ihn abgesenkt zu den restlichen Punkten – mit etwas Umrechnerei.



Das Ergebnis ist auf dem Ausgabebereich zu bewundern:



Benutzt wurden jeweils 300 „Personen“ ohne Multiplikatoren und mit nur einem anfänglich Infiziertem (rot: infiziert, gelb: immun, grün: gesund). Man sieht: soll in diesem Modell die Hälfte der Bevölkerung gesund bleiben, dann muss man schon 30% impfen.



3 Beispiele zu „Daten und Information“

3.1 Beispiele zur Kommunikation in gegebenem Kontext

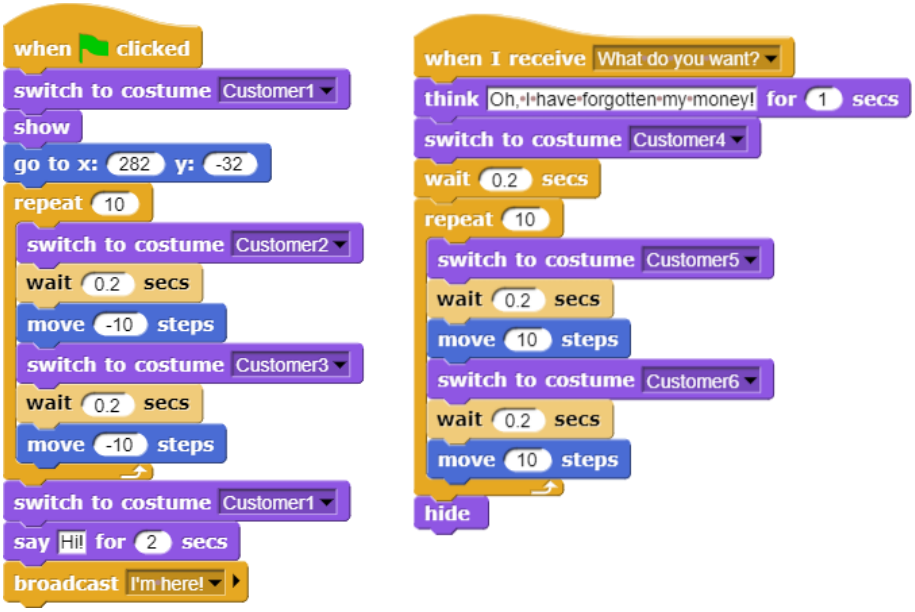
Wie bei den didaktischen Überlegungen beschrieben, benötigen wir Szenarien, in denen das Informatiksystem nur als Transportmittel für Nachrichten auftritt, die von den Beteiligten „verstanden“ werden. Selbst ist es damit erstmal nachrangig, im einfachsten Fall stellt es den Kontext nur dar, z. B. bei der Programmierung einer Geschichte. Das Informatiksystem ist aber trotzdem nicht irrelevant, weil einerseits dessen Gebrauch erlernt wird und somit spätere, eher „informatische“ Aufgaben vorbereitet werden. Andererseits stellt die Benutzung unterschiedlicher Objekte, die über Nachrichten kommunizieren, einen intuitiven Einstieg in die objektorientierte Modellierung dar. Die folgenden Beispiele eignen sich deshalb besonders für den Anfang eines Programmierkurses.

Im Gemüseladen

Altersstufe: *ab Sekundarstufe I* Material: *At the greengrocers*

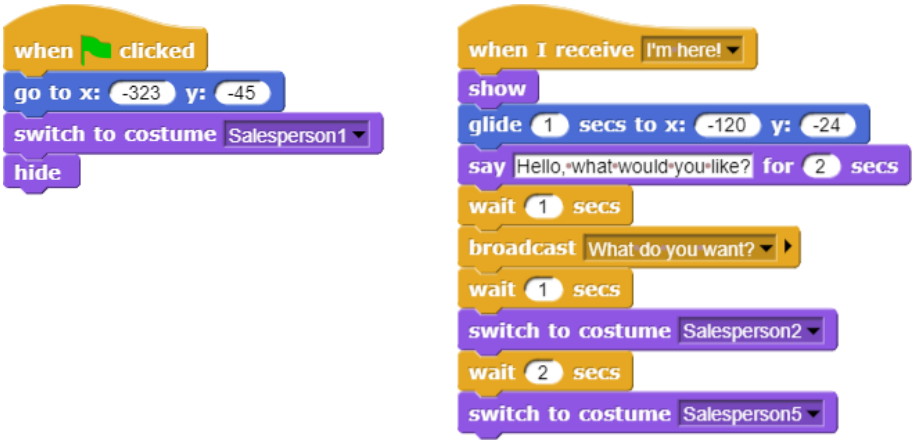
Zwei Personen agieren in einem Laden[25], z. B. indem eine Kundin den Raum betritt (mit Beinbewegungen durch Kostümwechsel) und anschließend eine Botschaft („Ich bin da!“) sendet. Daraufhin erscheint die Verkäuferin, fragt nach den Wünschen, … - alles durch Botschaften gesteuert. Der Kontext ist in diesem Fall eindeutig und weitgehend durch das Hintergrundbild gegeben, und da die Objekte nur auf bestimmte Botschaften reagieren, ist auch klar, was jeweils zu tun ist. Auch wenn die Situation trivial ist, besteht kein Zweifel an der Rollenverteilung: Botschaften im Informatiksystem bestehen aus Texten, die von den Handelnden interpretiert werden und ggf. Handlungen auslösen.

Skripte der Kundin:



[25] Kostüme teilweise aus Scratch und/oder eigenen Fotos.

Skripte der Verkäuferin:



Das gegebene Animationsprogramm, an dessen Erstellung die Lernenden beteiligt sein sollten, stellt einen einfachen Rahmen für den Anfangsunterricht bereit, der von den Lernenden modifiziert und ergänzt wird. Nicht zu unterschätzen ist dabei die Arbeit mit den Kostümen, z. B. um Bewegungen zu visualisieren. Der Umgang mit dem eingebauten Grafikeditor und anderen Grafikprogrammen, die dann wiederum unterschiedliche Grafikformate bereitstellen, was z. B. für die Transparenz des Hintergrunds wichtig ist, motiviert einen Teil der Lernenden mehr als der direkte Einstieg über Skripte. Sind verschiedene Kostüme erstellt, dann sollen sie natürlich auch benutzt werden – und dafür benötigt man dann doch Programmskripte. Der Umweg über die Grafik führt zur Algorithmik – allerdings anhand selbst erstellter (Teil-)Produkte, was die Motivation stark erhöhen kann.

Weshalb ist eigentlich die grafische Darstellung für die Lernenden so wichtig? Der Kontext erschließt sich eben nicht nur aus den Texten, sondern bei diesen einfachen Geschichten auch aus dem Aussehen, der Körperhaltung usw. der Akteure und aus deren Umfeld (siehe: „Im Bistro“ weiter unten). Was hier dargestellt wird, muss nicht mehr gesagt werden, ist aber entscheidend für die Interpretation. Es ist eben ein Unterschied, ob der Ausruf „Das ist ja Kohl!“ in einem Gemüseladen oder in einem Klassenzimmer erfolgt.

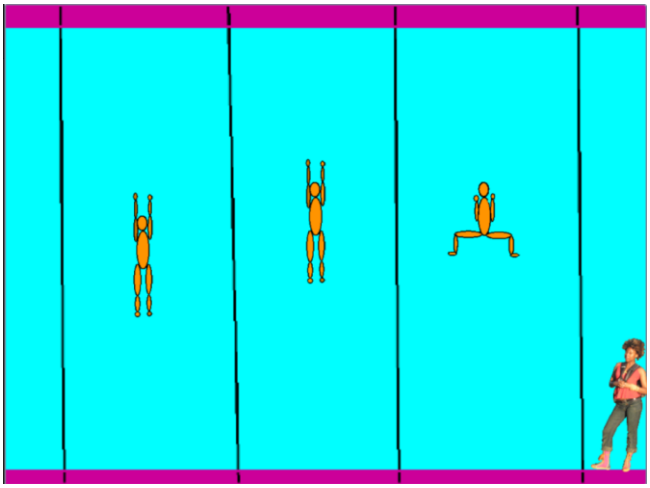
Das Beispiel führt zu unterschiedlichen Geschichten, die nach Herstellung eines definierten Anfangszustands („*grüne Flagge gedrückt*“) im Wechselspiel von gesendeten Botschaften und dadurch ausgelösten Ereignissen (mit deren Behandlung) in Anlehnung an die gegebenen Beispiele entstehen. Die Qualität der Ergebnisse zeigt sich dann in der Fantasie, Komplexität und Witzigkeit der Geschichten.

Obwohl Botschaften, Ereignisse, ggf. Zustände, die sich durch lokale Variable als „*zusätzliche Attribute zu den schon vorhandenen*“ beschreiben lassen, benutzt werden, sollte der Schwerpunkt des Geschehens nicht bei der Fachsprache liegen. Natürlich muss über die Vorgänge geredet werden, und dann kann man auch gleich die üblichen Begriffe benutzen. Das ist aber nur ein (durchaus wünschenswerter) Nebeneffekt. Ziel des Unterrichts ist es, die Lernenden zu aktivieren und fantasievolles Agieren zu fördern. Die Benutzung einer formalisierten Sprechweise („Das Attribut *x-Wert* des Objekts *Kundin* wird durch Aufruf der Methode *ändere x um -5* neu gesetzt.“) gehört eher nicht dazu. Es reicht festzustellen, dass die Kundin nach links geht.

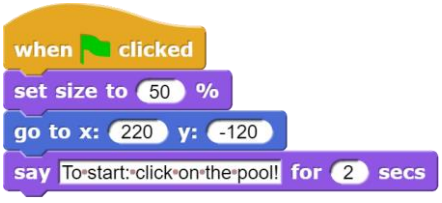
Schwimmer

Altersstufe: *ab Sekundarstufe I* Material: *Swimmer*

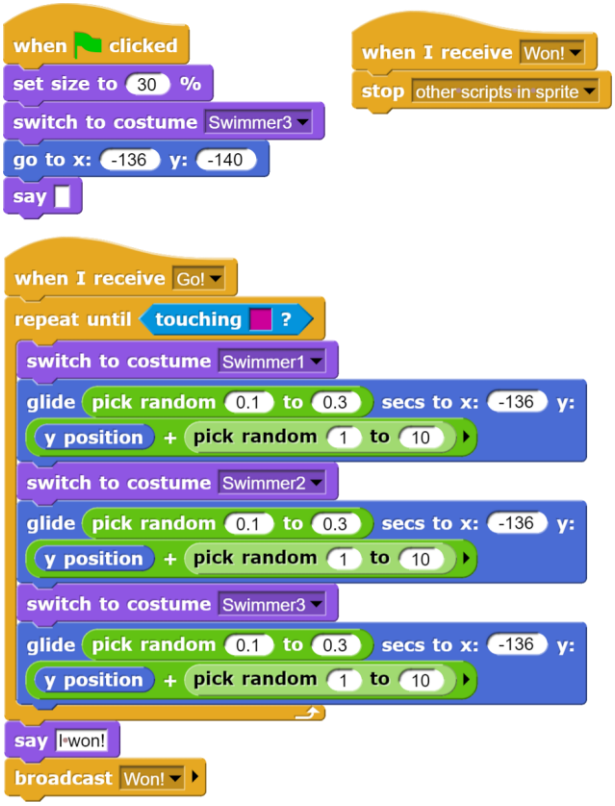
Wir zeichnen einen Schwimmer in verschiedenen Schwimmphasen und duplizieren ihn mehrmals. Auf eine Nachricht hin („Go!“) schwimmen die Schwimmer mit zufällig gewählten, anfangs veränderlichen Geschwindigkeiten bis zum anderen Ende der Schwimmbahn. Erreicht einer den Rand, dann freut er sich und stoppt durch eine Nachricht alle anderen Schwimmer (und sich selbst).



Die Miniskripte der Trainerin sind trivial …



… und auch die Schwimmer haben nicht viel mehr zu tun, als zu schwimmen.



Auch dieses Beispiel, dessen Hintergrundbild den Kontext ohne weitere Erklärungen definiert, dient nur dazu, die Unterschiede zwischen Botschaft und übermittelter Information deutlich zu machen. Es lässt sich allerdings leicht ausbauen, z. B., indem den Schwimmern feste Geschwindigkeiten zugewiesen werden (was ein neues Attribut, also lokale Variable erfordert) oder diese am Bahnende umkehren, um zum Startpunkt zurückzuschwimmen. Andere Schwimmstile sind leicht darstellbar, es lassen sich Erfolgsstatistiken führen und auch der Anschlag am Ziel ist stark verbesserungswürdig. In welchem Zusammenhang die Siegesnachricht mit der Äußerung der Trainerin „*Ich glaub‘ es nicht!*“ steht, bleibt allerdings das Geheimnis der Beteiligten.

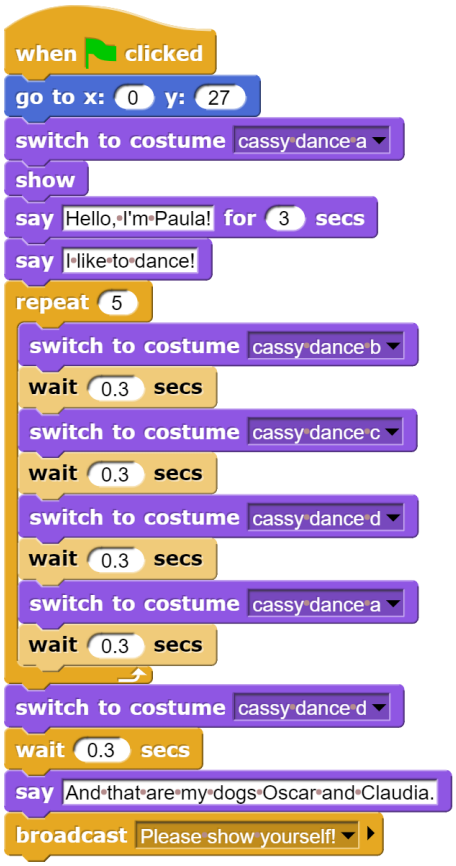
Selbstportrait

Altersstufe: *ab Sekundarstufe I* Material: *Self-portrait*

Die Schülerinnen und Schüler stellen sich selbst vor: wo sie wohnen, ihren Schulweg, ihre Hobbies, …

In diesem Fall ist die Rollenverteilung noch viel klarer: das Informatiksystem stellt Bilder und Texte, also Daten, bereit. Deren Auswahl oblag der portraitierten Person, und diese Auswahl soll Paula den Empfängern der Daten sympathisch erscheinen lassen. Klappt das?

Skript von Paula:



Paula weiß nichts über die Empfänger ihrer Daten. Falls diese Hunde mögen, wird es mit der Sympathie schon klappen: Hundewelpen finden fast alle süß. Falls diese aber eine Katze haben, die vom Nachbarhund gejagt wird, werden sie finden, dass Paulas Hunde in ein paar Monaten mit Vorsicht zu genießen sein werden. In diesem Fall sollte Paula lieber Bilder ebenso süßer Katzenkinder verwenden. Es wäre also in Paulas Interesse, möglichst viel über die Empfänger zu wissen und ihre Selbstdarstellung diesen Interessen anzupassen, wenn sie allen sympathisch erscheinen will.

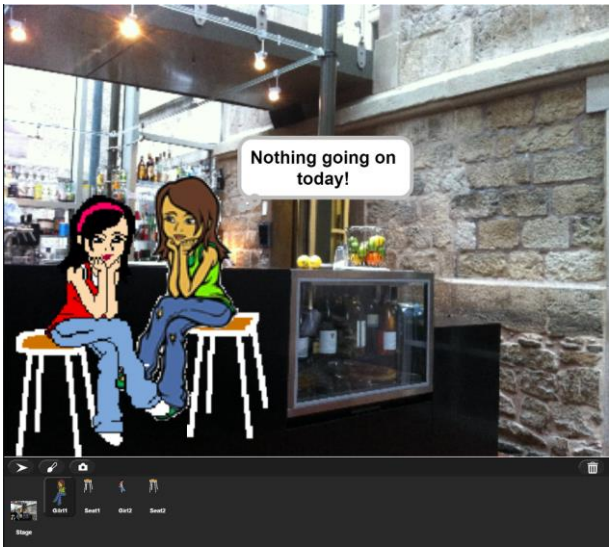
Wäre Paula ein Politiker, eine Firma, eine sonstige Institution, dann hätte sie ein massives Interesse, Daten ihrer „Leser“ zu gewinnen. Selbst Trivialdaten wie Hunde-/Katzen-Vorlieben lassen sich z. B. aus dem Kauf von Tierfutter erschließen – nicht immer richtig, aber meist. Für Normalbürger sind sie wahrscheinlich uninteressant, für den Politiker aber nicht, denn der muss entscheiden, ob er auf seinen Pressefotos mehr Kinder, Hunde oder Katzen herzt. Kann er jedem Interessenten das passende Bild schicken, dann wird auf der entstehenden Sympathiewelle auch einiges anderes an politischen Inhalten mitgeliefert werden können.

Wir können an diesem Beispiel sehen, dass schon ein triviales Projekt wie ein Selbstportrait Ansatzpunkte für die Diskussion gesellschaftlich relevanter Fragen liefert, hier: für die Motivation zum Sammeln personenbezogener Daten.

Skripte der Hunde:

Im Bistro

Altersstufe: *Sekundarstufe II* Material: *In the bistro*



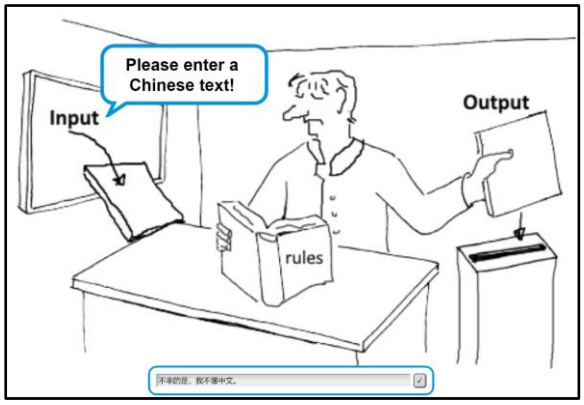
Das Beispiel entspricht direkt dem des Gemüseladens, vielleicht für etwas ältere Schülerinnen und Schüler. Die Nutzung von *Snap!* eröffnet einige Möglichkeiten, z. B. bei der Animation der Sprites. Man könnte z. B. die Gliedmaßen gesteuert am gemeinsamen Objekt bewegen („attached parts“). Vor allem aber wird durch die Nutzung von *Snap!* schon für einfache Animationen der Wechsel des Werkzeugs[26] überflüssig, wenn später komplexere Probleme bearbeitet werden. Und die Rolle der Körpersprache wird mehr als deutlich. Verlagert sich das Gespräch z. B. in den Bereich der Ironie, dann werden viele Äußerungen nur vor dem bildhaften Hintergrund richtig interpretierbar sein. Und natürlich: die Geschichte geht gerade erst los. Was passiert weiter? Man weiß es nicht!

[26] den ich immer für problematisch halte, weil dadurch der Werkzeuggebrauch als eigenes Problem zu sehr in den Vordergrund gerückt wird

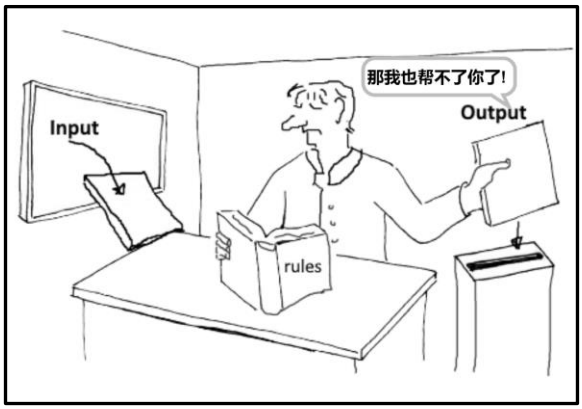
Searles chinesisches Zimmer

Altersstufe: *Sekundarstufe II* Material: *Chinese room*

Searls Beispiel dient zur Diskussion möglicher künstlicher Intelligenz. In einem Zimmer befindet sich eine Person, die kein Chinesisch kann, aber über ein Buch verfügt, das in ihrer Sprache Regeln zur Veränderung chinesischer Texte enthält. Sie bekommt chinesische Texte hereingereicht, wendet die Regeln an und gibt die Ergebnisse wieder nach außen.[27] Personen außerhalb des Zimmers glauben, dass die Person im Innern Chinesisch kann.



Das Beispiel ist angesichts der Diskussion über künstliche Intelligenz aktuell. Es ist aber natürlich auch ein exzellentes Beispiel zum Verhältnis von Information und Daten. Das „datenverarbeitende System“ im Innern versteht gar nichts, produziert aber Ergebnisse, die von den Benutzern als Erscheinungsformen von Intelligenz interpretiert werden.



Hinweis: Da ich leider kein Chinesisch kann, wurden im abgebildeten Beispiel die Texte von einem Programm übersetzt. Die Ergebnisse wurden dann in die Eingabefelder kopiert.

[27] https://de.wikipedia.org/wiki/Chinesisches_Zimmer

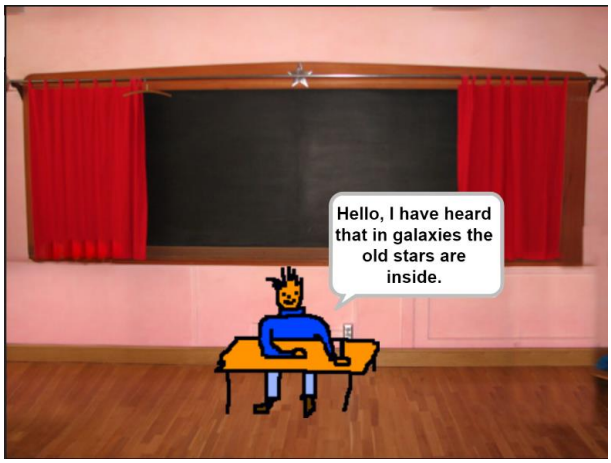
3.2 Beispiele zu Kommunikation mit offener Fragestellung

In diesem Szenario kommunizieren zwei menschliche Partner, die sich nicht kennen, mithilfe eines Informatiksystems. Der eine stellt eine Frage, der andere hilft ihm mit einer Antwort – hoffentlich nach bestem Wissen und Gewissen.

Fernunterricht Astrophysik

Altersstufe: *Sekundarstufe II* Material: *Distance learning*

Wir nutzen die Möglichkeit von *Snap!* aus, mit mehreren Bühnen zu arbeiten und damit zwei Projekte zu verwalten, die miteinander kommunizieren können. Ein Schüler stellt eine Frage an den weit entfernten Astrophysiker, der liefert ihm Material[28], von dem er hofft, dass der Schüler sich die Antwort erschließen kann. In diesem Fall handelt es sich dabei um einige Galaxienbilder. Durch die Frage wird ein Kontext erzeugt, für den der Antwortende Daten zusammenstellt und übermittelt, von denen er glaubt, dass sich dem Fragenden daraus die gesuchten Informationen erschließen. Der Fragende interpretiert dann das Datenmaterial als Hilfestellung bezogen auf seine Frage – und nicht etwa als Deko-Material für den Klassenraum.



Die Partner kommunizieren über den Block zum Wechseln der Szene und können sich dabei Texte oder andere Daten übermitteln.

Bei etwas komplizierteren Fragen müssen die Daten natürlich zuerst zusammengestellt, ausgewertet, dargestellt und interpretiert werden, bevor entschieden wird, ob die ursprüngliche Frage sich damit beantworten lässt (s. nächstes Beispiel). Auch darüber kann die Kommunikation mit dem Lehrenden erfolgen.

[28] Aus https://en.wikipedia.org/wiki/Galaxy

Die Skripte innerhalb der Projekte der Beteiligten sind einfach. Zu verstehen ist das Zusammenstellen der Bilddaten für die Übermittlung seitens des Astronomen und das inverse Erzeugen von Bildern seitens des Schülers, die Kommunikation zwischen den Szenen. Zentral ist somit der Datenaustausch zwischen entfernten Partnern.

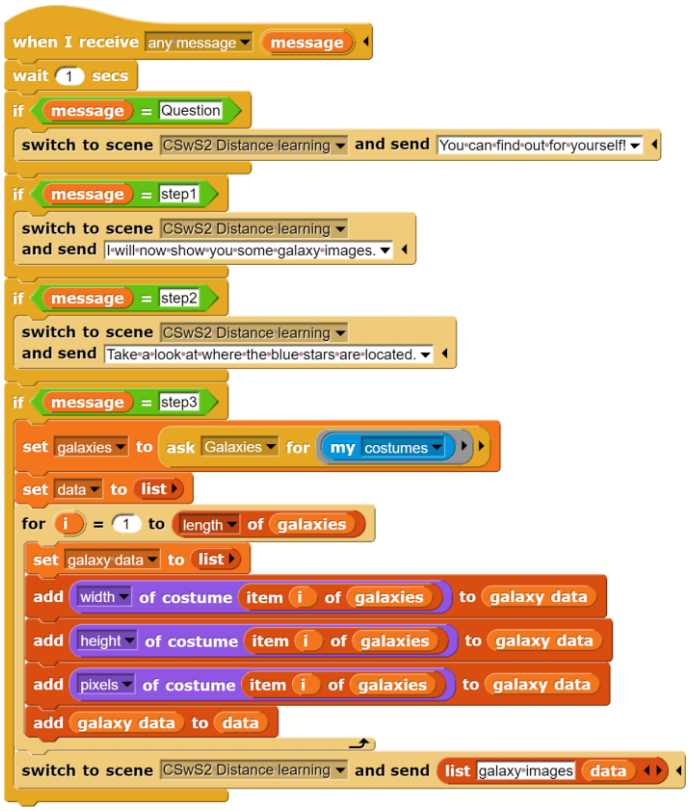
Der Schüler stellt seine Anfangsfrage. Dann veranlasst er, dass zu der Szene des Astronomen gewechselt wird.

Erhält er eine Antwort, dann sieht er zuerst einmal nach, ob es sich um eine Liste (also Daten) handelt. Ist das der Fall, dann baut er aus den Daten die Galaxienbilder zusammen und stellt sie auf dem Projektor in einer Endlosschleife dar. Er muss dazu natürlich wissen, wie die Daten strukturiert sind.

Erhält er keine Liste vom Astronomen, dann stellt er dessen Antwort 3 Sekunden lang dar und wechselt zurück zum Astronomen. Dabei zählt er die Dialogschritte durch (*step1, step2, …*).

Der Astronom möchte, dass der Schüler die Antwort auf seine Frage selbst findet. Er zeigt das schrittweise durch einige Anmerkungen, die er an die Szene des Schülers übermittelt.

Danach erzeugt danach aus den Galaxienbildern übermittelbare Daten, also Listen aus Breite, Höhe und Pixeln des Bildes, die er als Liste *data* zusammenfasst. Dann schickt er das ganze Paket an den Schüler.

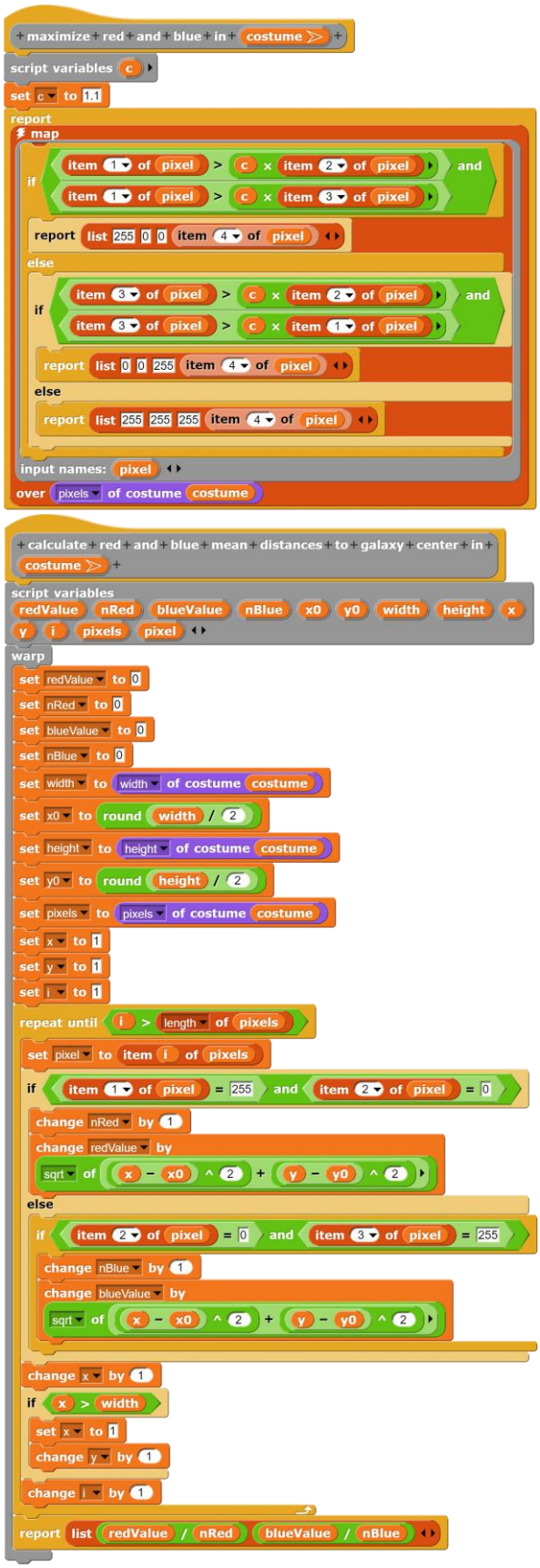


Das Vorgehen des Astronomen ist nur zu verstehen, wenn er den Fragen des Schülers entnimmt, dass dieser als Neuling zu eigenen Erkenntnissen angeleitet werden sollte. Ansonsten könnte er ja einfach mit „*ja*“ antworten. Er vermutet also keinen Kollegen, Journalisten, der für einen Artikel recherchiert, Tapetendesigner, der Bilder sucht, … an der anderen Seite der Leitung. Das kann stimmen – oder auch nicht. Sollte es nicht stimmen, dann kann der falsch interpretierte Kontext zu einigem Ärger führen. Beispiele dafür lassen sich leicht finden.

Berechnung des Abstands der roten bzw. blauen Pixel vom Zentrum der Galaxis

Altersstufe: *Sekundarstufe II* Material: *Distance learning II*

Wir wollen unser Astronomie-Beispiel fortsetzen und den Schüler in die Lage versetzen, die mittleren Abstände der roten bzw. blauen Pixel vom Zentrum der Galaxis zu messen. Dafür benötigen wir natürlich ein Werkzeug, das einerseits überhaupt RGB-Werte lesen und wieder schreiben kann, und das dann die aufbereiteten Bilddaten auswertet. Das ist ein typischer Auftrag an den Fachmann, unseren Astronomen. Der schreibt zwei Funktionen, die einmal die „überwiegend“ roten bzw. blauen Pixel aus einem Galaxienbild heraussuchen und verstärken und dann aus diesen Bildern die mittleren Abstände der roten bzw. blauen Pixel zur Bildmitte, also in etwa zur Mitte der Galaxie, berechnen. Diese Funktionen schickt er „per Mail“, d. h. über Dateiexport und -import an den Schüler.[29]



Man kann unterschiedlicher Meinung darüber sein, was man unter „überwiegend“ rot oder blau versteht. Die Version unseres Astronomen für *maximize red and blue* lautet:

Wenn der Rotwert (item 1) eines Pixels sowohl größer als der Grünwert (item 2) und der Blauwert (item 3) (incl. eines Faktors) ist, dann gib ein Pixel in vollem Rot zurück, entsprechend ein blaues Pixel, und sonst Weiß.

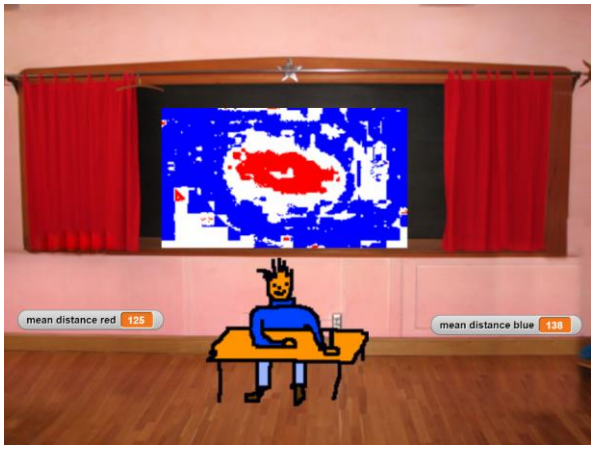
Benutzt wird eine der höheren Funktionen von *Snap!* (*map … over …*) in vorkompilierter Form. Das Ergebnis wird so sehr schnell ermittelt.

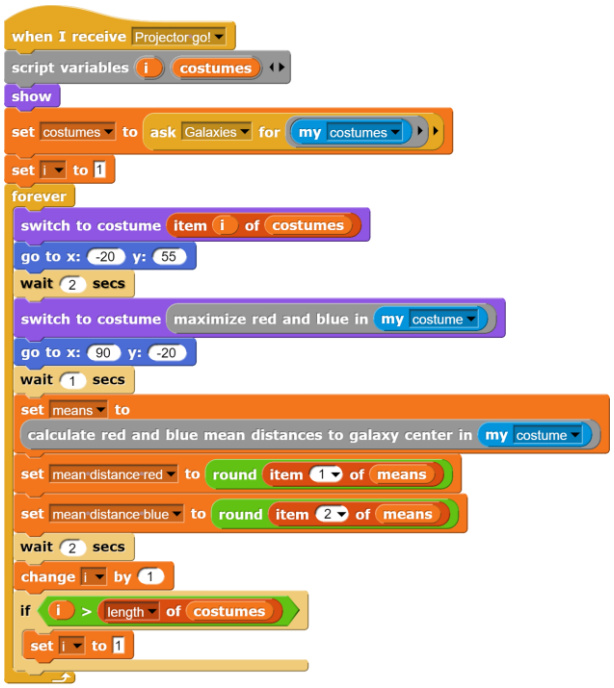
Die Funktion *calculate red and blue mean distances …* bestimmt zuerst einige Anfangswerte sowie die Bild-Dimensionen und die Pixel des Bildes. Dann berechnet sie für alle Pixel des Bildes die Abstände zur Mitte. Sie gibt als Ergebnis eine Liste mit den beiden Mittelwerten zurück.

Diese beiden Funktionen übermittelt der Astronom dem Schüler „per Mail“.

[29] Weil Snap! derzeit noch keine Skripte direkt zwischen den Szenen austauschen kann.

Unser Schüler bittet nach Erhalt der Mail den Projektor, die beiden Operationen auf alle Galaxienbilder anzuwenden und die Ergebnisse dazustellen. das tut der dann auch.





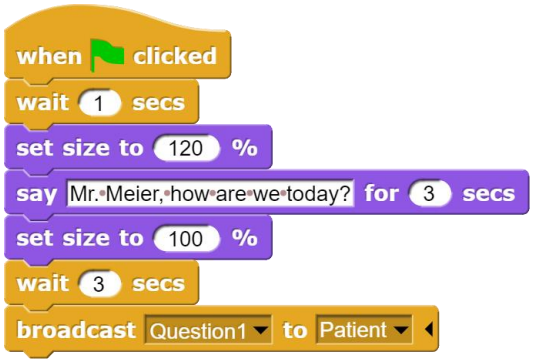
Soweit der informatische Teil.

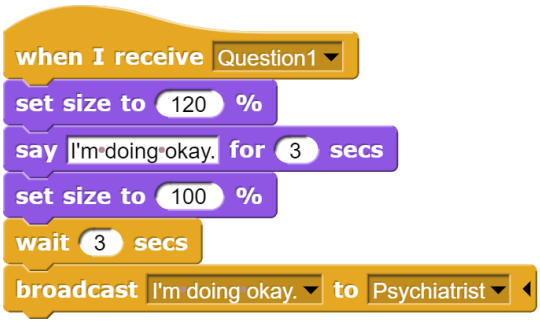
Erinnern wir uns daran, dass der Astronom die Bilder mit dem kleinen zusätzlichen Tipp als Antwort auf die Frage nach den alten Sternen geschickt hat, dann wurde diese Frage bisher nicht beantwortet. Der Schüler hat zwar zuerst per Augenschein, dann bestätigt durch ein kleines Programm festgestellt, dass im Innern der Galaxien mehr rot als blau strahlende Sterne zu finden sind - aber das wollte er doch gar nicht wissen. Er kann jetzt auf (mindestens) zwei Arten auf die Situation reagieren: entweder hält er den Astronomen für einen unfähigen Lehrer, der ihn und seine Fragen nicht ernst nimmt, und verlässt beleidigt das Fernunterrichtsprogramm, oder er vertraut dem Astronomen und schließt daraus, dass die alten Sterne die roten sind. Dieser zusätzliche Schluss hat aber nur teilweise mit den übermittelten Daten zu tun, Fakten dazu fehlen völlig. Er ergibt sich wesentlich aus dem Kontext und der Situation der beteiligten Personen.

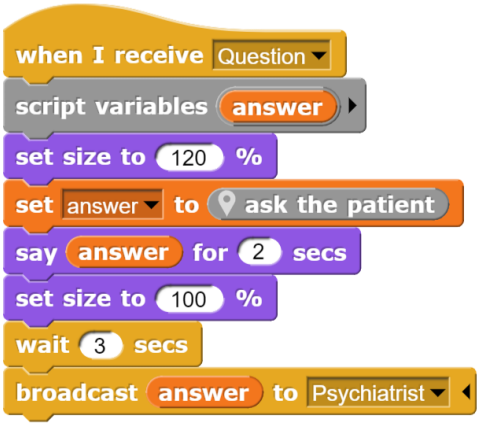
Weizenbaums Eliza[30]

Altersstufe: *Sekundarstufe II* Material: *Eliza*

Das berühmte Beispiel beschreibt die Kommunikation zwischen Psychiater und Patient, wobei (in diesem Fall) beide zufallsgesteuert Platituden absondern. Die Koordination des „Gesprächs“ geschieht wieder durch entsprechende Botschaften.







Neben dem spielerischen Charakter des Beispiels ist auch der Informationsgehalt der Nachrichten interessant. Patient und Arzt reagieren inhaltlich überhaupt nicht aufeinander, sie senden aber Daten zum richtigen Zeitpunkt. Was ist denn nun die übermittelte Information? Wenn überhaupt etwas, dann erfährt der Patient, dass jemand da ist, mit dem er reden kann. Vielleicht hilft ihm das. Diese Information wird aber nicht durch die übermittelten Daten repräsentiert, sondern durch die Anwesenheit von Daten(müll). Die Daten selbst sind irrelevant. Oder anders gesagt: es können beliebige Daten übertragen werden, denn nicht sie tragen die Information, sondern der Kontext gibt ihnen allen dieselbe Bedeutung.

[30] https://de.wikipedia.org/wiki/ELIZA

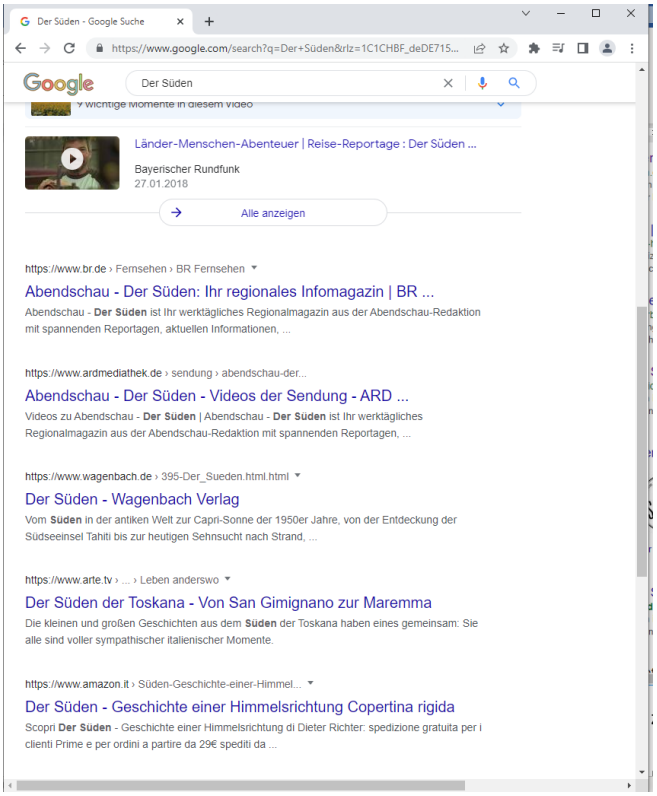
Reporter zur Bestimmung der Zufallsantworten von Psychiater und Patient:

3.3 Kommunikation mit eindeutiger Fragestellung

In diesem Szenario kommuniziert ein menschlicher Partner mit einem Informatiksystem. Will er angemessene Antworten haben, dann muss er seine Fragen entsprechend formulieren.

Die Wissensgesellschaft

Altersstufe: *Sekundarstufe II*
Material: *Ergebnisse der Recherchen*[31]



Man hört oft, dass es nicht mehr nötig sei, Wissen zu erwerben, weil sich dieses ja „im Netz“ befinde und jederzeit abgerufen werden könne.

Ist das so?

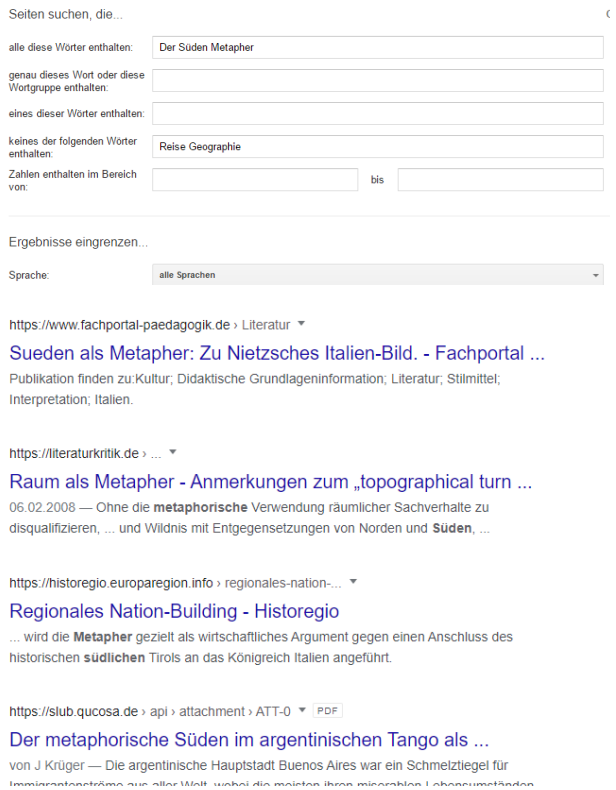
Wir gehen wie immer experimentell vor und versuchen, uns über einen nicht ganz trivialen Begriff zu informieren: den Süden. Die Antwort bilden die „relevantesten“ der 47 400 000 Ergebnisse. Geliefert werden die üblichen Wikipedia-Einträge z. B. zu dem Buch von Borges („*El Sur*“), zu Fernsehsendungen, Filmen und der Himmelsrichtung sowie Hinweise auf Reiseliteratur – und auf weitere Bücher zum Thema. Auf den folgenden Seiten finden wir auch nicht viel mehr. Wir müssen also daraus schließen, dass „der Süden“ allein geografisch zu verstehen ist – oder wir müssen in den bitteren Apfel beißen und richtige Bücher lesen. Einige wenige davon scheint es nach Googles Meinung ja zu geben. Lehnen wir dieses üble Ansinnen ab, dann bleibt es beim geografischen Süden. Hinweise auf den Süden als Metapher, soziales oder ökonomisches Phänomen, narratives Element, Sehnsuchtsort, literarische Kategorie, Thema der bildenden Kunst usw. fehlen, und wir werden sie nicht einmal vermissen, es sei denn, wir wüssten, dass es sie gibt.

„Im Netz“ finden wir Fakten: die Einwohnerzahl von Hamburg oder das Bruttosozialprodukt von Burkina Faso, das Rezept für Frutti di Mare oder zur Reparatur des Staubsaugers. Aus diesen Fakten lassen sich Informationen gewinnen, wenn wir sie geeignet auswerten und einordnen. Doch worin „einordnen“? Infrage dafür kommt nur vorhandenes Wissen, im Kopf vorhandenes, und das muss erst einmal erarbeitet werden, bevor „das Netz“ angemessen benutzbar ist.

Fragen wir nach den Ergebnissen solcher Ordnungsprozesse, also den ausgewerteten Daten, dann erhalten wir natürlich auch Antworten: die Meinungen anderer. Diese können wir aber nur bewerten, also selbst wieder einordnen, wenn wir über entsprechende Fähigkeiten verfügen (s. o.). Fehlen diese, dann bleiben andere Bewertungskriterien: dass wir den Meinungsbildnern glauben (oder nicht), dass sie uns sympathisch sind (oder nicht), dass sie sind wie wir (oder nicht), dass andere ihnen glauben (oder nicht) … – wenn wir glauben, dass die anderen die sind, die sie angeben zu sein (oder nicht). Mit Rationalität hat das wenig zu tun.

[31] Die abgebildeten Screenshots stammen aus Google Chrome vom 1.5.2022

Wenn wir wissen, dass es noch andere Möglichkeiten gibt, unsere Fragen zu beantworten, als die zuerst von der Suchmaschine gelieferten, dann sind wir fein raus. Erweitern wir unsere Suche nach *„dem Süden“* um den Begriff *„Metapher“*, dann erhalten wir ein völlig anderes Spektrum von Antworten – und es gibt nur noch 200 000 Ergebnisse. Welche Verarmung! Selbst *„der Süden als Sehnsuchtsraum“* ergibt 609 Antworten, die fast nichts mit den vorherigen gemein haben. Erst die Kombination mit der bildenden Kunst liefert wieder einige hunderttausend Treffer. Präzisieren wir unsere Anfrage, indem wir erweiterte Einstellungsmöglichkeiten benutzen oder schon wissen, wie man z. B. Begriffe ausschließt, dann kommen die Suchergebnisse langsam dem näher, was wir von ihnen erwarteten. Auch hier liefert vorhandenes Wissen den Zugang zu neuem.

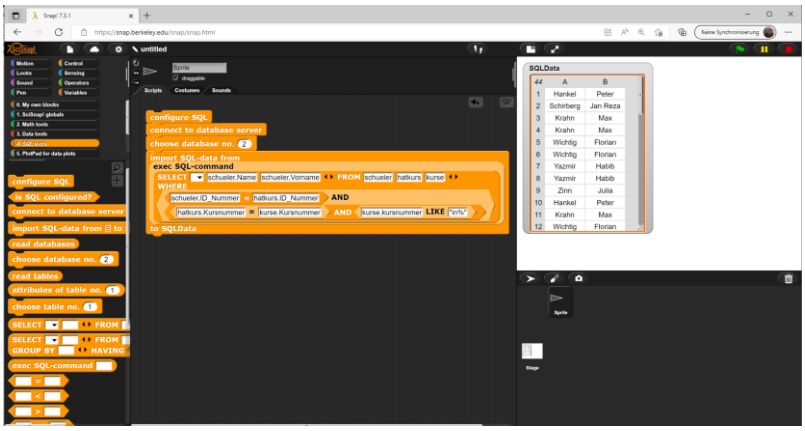


Suchmaschinen sind ja nicht gehässig, sie arbeiten nur „wie vorgegeben“. Stellen wir präzise Fragen, dann liefern sie meist auch präzise Antworten. Stellen wir keine präzisen Fragen, dann benötigen sie Zusatzkriterien, um „die besten“ Antworten zu finden. Bei diesen Kriterien kann es sich um bezahlte Platzierungen handeln, meist aber um die „Bewertung“ der Antworten durch andere Benutzer, die gleiche oder ähnliche Suchanfragen gestellt haben, oder andere gespeicherte Daten. Die Bewertung findet bekanntlich in Form eines Klicks auf die Antwortzeile statt.

Dieses Verhalten hat Konsequenzen. Niemand kann die anfangs genannten 47 400 000 Antworten durchforsten, und selbst die 609 Treffer des Sehnsuchtsraums sind fast unüberschaubar. Also werden fast alle Klicks auf den ersten ein, zwei Seiten der Suchergebnisse erfolgen – und damit haben wir einen selbstverstärkenden Prozess: die meistangeklickten Seiten werden wieder am meisten angeklickt, womit sich ihre Platzierung weiter festigt. Die anderen sind zwar vorhanden, aber praktisch unsichtbar. „Im Netz“ erscheinen den Benutzern dauerhaft nur Seiten, die inhaltlich denen entsprechen, die ihnen anfangs angeboten wurden. Neue werden kaum dazukommen. Wurden z. B. die ersten Suchanfragen vom Anbieter gefiltert, ihm die Ergebnisse „ähnlicher“ Benutzer geliefert, die sich aufgrund seiner bisherigen Nutzung des Systems (oder zunehmend „des Netzes“ als Ganzem) leicht finden lassen, dann wird der Benutzer diesen „Informationsraum“ kaum wieder verlassen. Er sieht einfach nichts Anderes. Seitens des Anbieters ist dieses Verhalten verständlich, denn der möchte Suchergebnisse liefern, die mit hoher Wahrscheinlichkeit angeklickt werden - nur dann wird er bezahlt. Aber die so entstehenden „Echokammern“ sind z. B. politisch brandgefährlich, weil sie die Gesellschaft in disjunkte Gruppen spalten, die kaum noch diskursfähig sind, aber auch das Spektrum der sonst gelieferten „Informationen“ verarmen.

Das Resultat unserer Überlegungen ist ziemlich eindeutig: „Das Netz“ enthält kein Wissen, sondern Daten. Diese können unser Wissen bereichern, wenn wir über das Wissen verfügen, sie angemessen zu nutzen, sie zu bewerten, sie einzuordnen oder zu verwerfen. Eine entsprechende Bildung bildet die Basis für wunderbare neue Möglichkeiten. Fehlt sie, dann werden wir zu manipulierbaren Objekten.

Zugriff auf Datenbanken



Altersstufe: *Sekundarstufe I/II*
Werkzeug: *SciSnap!*[32]
Material: SQL example

Wir benutzen eine der Bibliotheken von Snap! (SciSnap!), die u. a. die Möglichkeit bietet, auf Datenbanken zuzugreifen. Damit können wir die üblichen Datenbankanfragen aus den zugehörigen Relationen, Attributen usw. mithilfe von Blöcken zusammensetzen und ausführen lassen. Das Ergebnis ist jeweils eine Liste. Im gezeigten Beispiel erhalten wir die Teilnehmenden an Informatik-Grundkursen. Ganz einfach.

Warum ist das so einfach – und für wen? Der Benutzer muss die SQL-Syntax kennen und sich an sie halten. Vor allem aber muss er die gewünschten Daten völlig eindeutig beschreiben, es darf keine Interpretationsmöglichkeit geben. Das ist gar nicht so einfach. Die Auswertung solcher Anfragen ist aber dann für die Maschinen einfach, das können sie schon seit Jahren.

Unser Benutzer beschreibt mit seiner Anfrage die Daten, die er vom System erhalten möchte. Er weiß zwar meist nicht, welche Daten genau er dann erhalten wird, er kennt aber ihre Bedeutung. ER kennt sie, nicht die Maschine. Die kann sie gar nicht kennen, weil sie nicht wissen kann, welche Bedeutung der Benutzer diesen Daten gibt, welche Information er ihnen entnehmen wird. Die Liste unser Kursteilnehmer z. B. kann viel bedeuten: vielleicht müssen sie wegen einer Exkursion vom restlichen Unterricht des Tages befreit werden, vielleicht wird nachgesehen, welche kleinen Kurse man streichen kann, vielleicht wird überprüft, ob die Bücher für den Kurs reichen. Man weiß es nicht …

SQL-Anfragen sind für Maschinen einfach auswertbar, weil sie eine klare Entscheidungsgrundlage liefern: Daten gehören entweder zur angeforderten Kategorie – oder nicht. Interessant wird es, wenn Fragen gestellt sind, die keine eindeutigen Entscheidungshilfen liefern. Sollen solche unscharfen Fragestellungen wie z. B. „*Werden die Jungen (oder die Mädchen) in der Schule benachteiligt?*“, „*Haben es Stadtkinder (Landkinder, Kinder aus bürgerlichen Elternhäusern, Mitgliedern in Sportvereinen, Kindergartenkinder, Kinder mit Migrationshintergrund, …) in der Schule leichter (schwerer)?*“ „*Ist die Bewertung unterschiedlicher Kompetenzen in der Schule gerecht?*“ mithilfe von (z. B.) SQL-Anfragen beantwortet werden, dann muss zwangsläufig eine Umformulierung und damit eine Interpretation erfolgen, die zu Antworten führt, bei denen zumindest fraglich ist, ob sie die ursprüngliche Frage beantworten oder eben nur die Interpretation. Eine Antwort gibt es immer, wenn eine Anfrage formuliert wurde, auch dann, wenn die ursprüngliche Frage aufgrund der Daten eigentlich nicht zu beantworten ist.

[32] SciSnap! ist eine der Snap!-Bibliotheken.

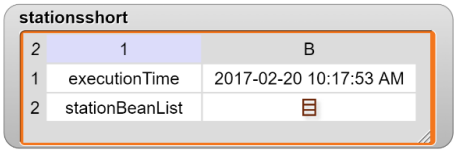
Zugriff auf JSON[33]-Daten

Altersstufe: *Sekundarstufe II*
Material: *JSON example*
stations.json (enthält die Daten von Fahrrad-Entleihstationen in New York)
stationsshort.json (gekürzte Version der Datei)

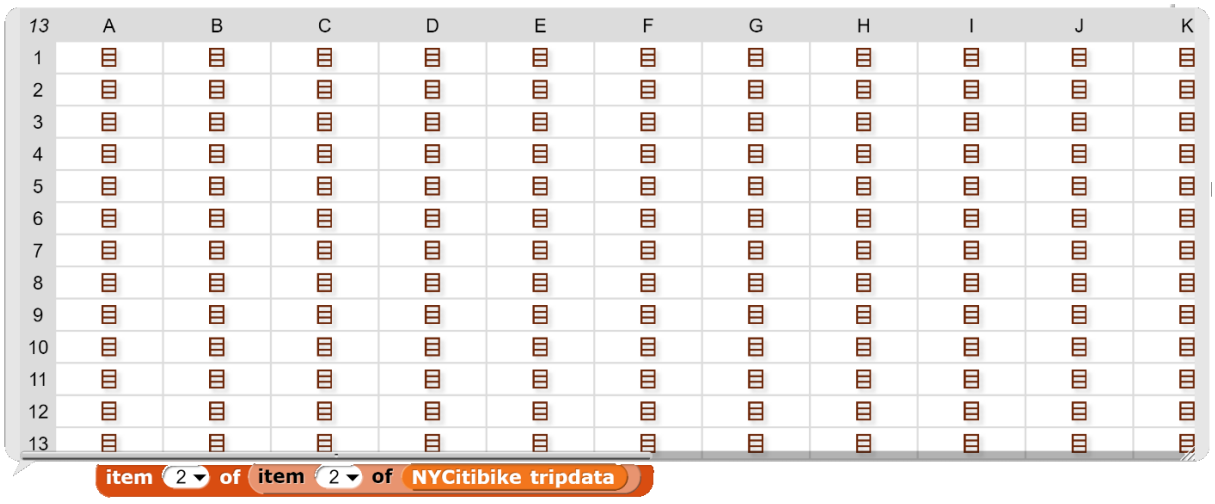
Bei JSON handelt es sich um eine Zeichenketten-Datenstruktur, die als Datei gespeichert wird. *Snap!* kann solche Dateien importieren und in eine geschachtelte Liste umwandeln, sowie ggf. mithilfe des *<length> of <list>* - Blocks Listen in das JSON-Format umwandeln und wieder exportieren. Wir laden also die JSON-Daten auf unseren Rechner und importieren sie direkt in eine Variable, die wir vorher angelegt haben. Noch einfacher geht es, wenn wir die JSON-Datei einfach auf das *Snap!*-Fenster ziehen. In diesem Fall wird eine Variable mit dem Dateinamen angelegt und mit den Dateiinhalten gefüllt.

Im Projekt sollen aktuelle und frei zugängliche Daten, die als JSON-Dateien gespeichert wurden, ausgewertet werden. Dazu müssen die Lernenden erst einmal recherchieren, um was es sich bei JSON handelt, welche Struktur die Dateien haben, welche Größen in ihnen darstellbar sind. Als Beispiel wählen wir die Daten der Fahrrad-Entleihstationen in New York, die in einer Datei *stations.json* vorliegen.[34] Als Ziel der Transformation dient in diesem Fall eine Struktur, die entweder eine atomare Größe (Wahrheitswert, Zahl, Zeichenkette, …) oder eine Liste enthält, die aus atomaren Größen und/oder Teillisten besteht, die als ersten Eintrag den Typ der originalen Daten (Liste oder Dictionary) enthalten. In Dictionaries folgen als weitere Elemente zweielementige Listen mit Schlüssel/Wert-Paaren.

In unserem Fall sieht das Ergebnis des Datenimports etwas enttäuschend aus:



Wir sehen uns deshalb das zweite Element der zweiten Zeile der Liste etwas genauer an – da scheint sich ja etwas zu verbergen.



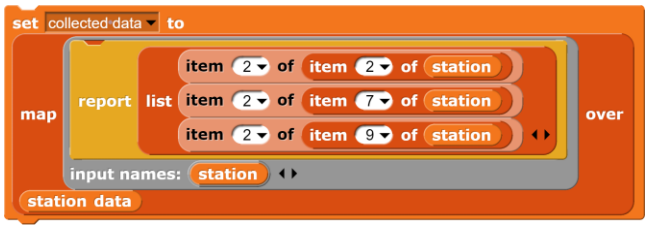
[33] JavaScript Object Notation
[34] https://catalog.data.gov/dataset/citi-bike-live-station-feed-json-d1c27

Das sieht schon interessanter aus. Mal sehen, was die erste Zeile enthält:

18	A	B
1	id	72
2	stationName	W 52 St & 11
3	availableDoc	25
4	totalDocks	39
5	latitude	40.76727216
6	longitude	-73.9939288
7	statusValue	In Service
8	statusKey	1
9	availableBike	13
10	stAddress1	W 52 St & 11
11	stAddress2	
12	city	
13	postalCode	
14	location	
15	altitude	
16	testStation	false
17	lastCommun	2017-02-20
18	landMark	

Das sind also die gesuchten Stationsdaten. Wir speichern deshalb diese Elemente in einer neuen Variablen.

Aus diesen Daten soll nun eine Tabelle erzeugt werden, die nur die Spalten *Stationsname*, *Status* und *verfügbare Fahrräder* enthält.



collected data

13	A	B	C
1	W 52 St & 11 Ave	In Service	13
2	Franklin St & W Broadway	In Service	6
3	St James Pl & Pearl St	In Service	13
4	Atlantic Ave & Fort Greene Pl	In Service	17
5	W 17 St & 8 Ave	In Service	4
6	Park Ave & St Edwards St	In Service	6
7	Lexington Ave & Classon Ave	In Service	2
8	Barrow St & Hudson St	In Service	19
9	MacDougal St & Prince St	In Service	6
10	E 56 St & Madison Ave	Not In Service	0
11	Clinton St & Joralemon St	In Service	5
12	Nassau St & Navy St	In Service	12
13	Hudson St & Reade St	In Service	0

Liegen die Daten wie hier in Listenform vor, dann kann ihr relevanter Teil leicht extrahiert und ausgewertet werden – bloß, was ist hier „relevant“. Das hängt natürlich davon ab, was mit den Daten geschehen soll, welche Informationen gesucht sind. Interessiert uns die Zahl der verfügbaren Räder im Verlauf der Woche, dann ergibt sich eine andere Auswertung als beim Wunsch, die Verteilung der Räder über die Stadt in einem Stadtplan darzustellen. Und vielleicht suchen wir ja auch nur ein freies Rad in der Nähe des Hotels.

Und hätten wir das Ganze nicht einfach einer SQL-Abfrage überlassen können? Das hätten wir, wenn die Daten in einer SQL-Datenbank vorliegen würden. Das tun sie in diesem Fall aber nicht. Statt also eine genaue SQL-Beschreibung der gesuchten Daten zu geben und deren Auswertung dem SQL-Server zu überlassen, werden wir in diesem Fall tätig, indem wir die genaue Beschreibung algorithmisch formulieren. Das Ergebnis ist das gleiche. In jedem Fall obliegt es dem menschlichen Benutzer, seine Wünsche so exakt zu beschreiben, dass die Maschine eine eindeutige Anweisungsfolge erhält, die sie zur Zusammenstellung der Antwort benötigt.

3.4 Kommunikation ohne menschliche Partner

Für dieses Szenario benötigen wir Beispiele, in denen Daten von einem System erfasst, an ein anderes, das durchaus im gleichen Rechner laufen kann, übermittelt und dort ausgewertet werden. Die Resultate dieser Auswertung werden dann diskutiert.

Nummernschilderkennung

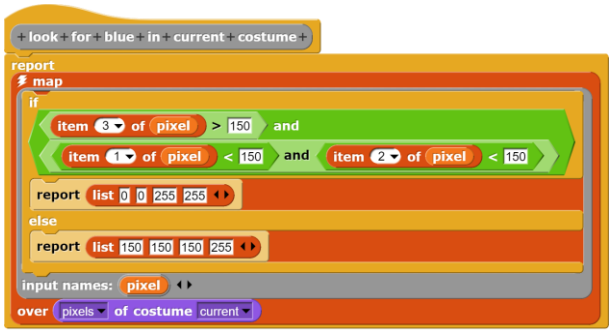
Altersstufe: *Sekundarstufe I/II* Material: *License plate detection*

Wir wollen uns mit dem weiten Feld der Zeichenerkennung beschäftigen, d. h. aus einem Bild Texte entnehmen. Als Beispiel wählen wir die Nummernschilderkennung, wie sie z. B. in den Mautbarrieren an Autobahnen praktiziert wird. Wir wählen hier den einfachen, für die Mittelstufe geeigneten Fall, dass wir uns nur für die Nationalitäten der Fahrzeuge interessieren.[35] Damit müssen wir nur die Zeichen im blauen Bereich des Europa-Nummernschilds erkennen. Bilder für solche Aufgaben lassen sich schnell und einfach im Internet generieren.[36]

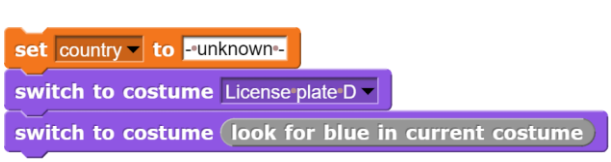
Wir wählen einen sehr einfachen Ansatz und hoffen, dass sich die Anzahlen der Pixel zur Darstellung dieser Zeichen unterscheiden. Damit reduziert sich das Problem auf die Aufgabe, den blauen Bereich zu finden und in diesem die nichtblauen Pixel zu zählen. Um unabhängig von der Größe der Darstellung zu sein, vergleichen wir die Pixel im oberen Bereich („Euro-Sterne“) mit denen im unteren. Der Einfachheit halber arbeiten wir zum Datentransport zwischen den verschiedenen Blöcken mit globalen Variablen.

Zuerst einmal generieren wir einige Nummernschilder unterschiedlicher Nationalitäten, importieren diese als Kostüme und schreiben einige Methoden, die die Teilaufgaben lösen.

Wir müssen zumindest den blauen Bereich im Kennzeichen finden. Ähnliches haben wir z. B. bei den Galaxienbilder schon gemacht. Bei den RGB-Grenzwerten muss man etwas experimentieren, dann klappt es gut. Da ein Bild ziemlich viele Pixel hat, arbeiten wir mit der kompilierten Version von *map…over*. Dann geht die Erkennung sehr schnell.



Bisher haben wir dann das folgende Teilergebnis.

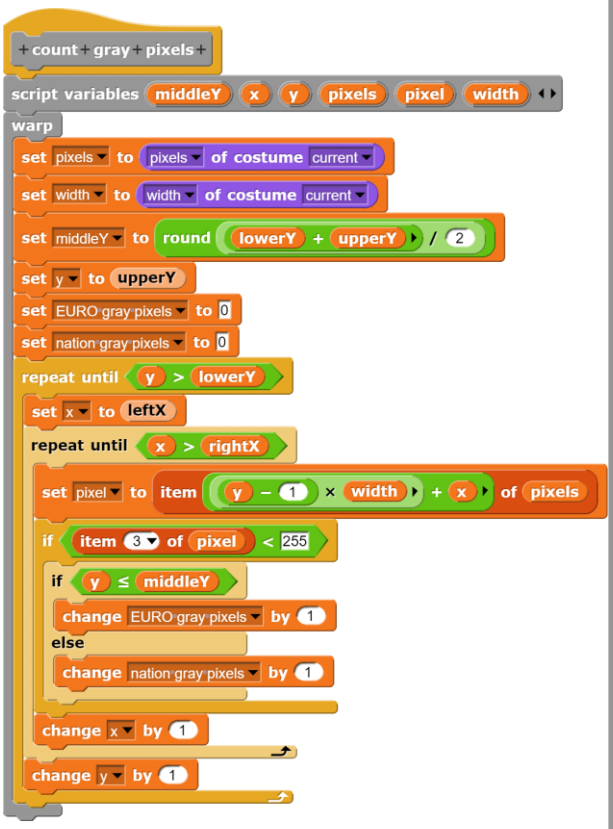


[35] Ein ausführlicheres Beispiel finden wir auf http://snapextensions.uni-goettingen.de/beispielsupermarkt.pdf. Dort werden neben der Zeichenerkennung auch einfache Ansätze zur Gesichtserkennung usw. realisiert.

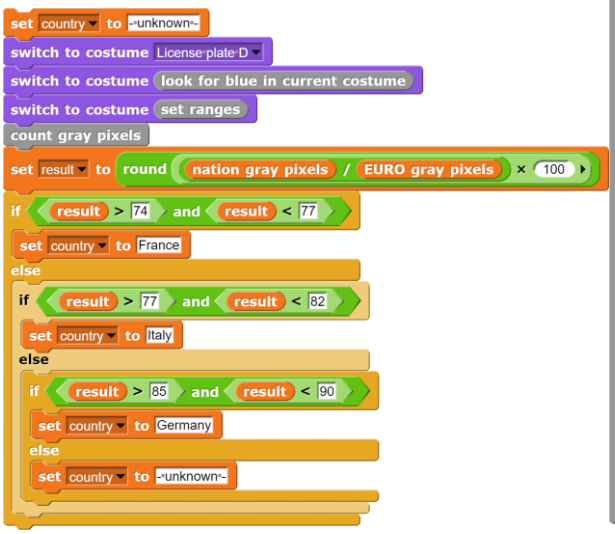
[36] Man kann z. B. einfach unter dem Stichwort „Nummernschild“ suchen.

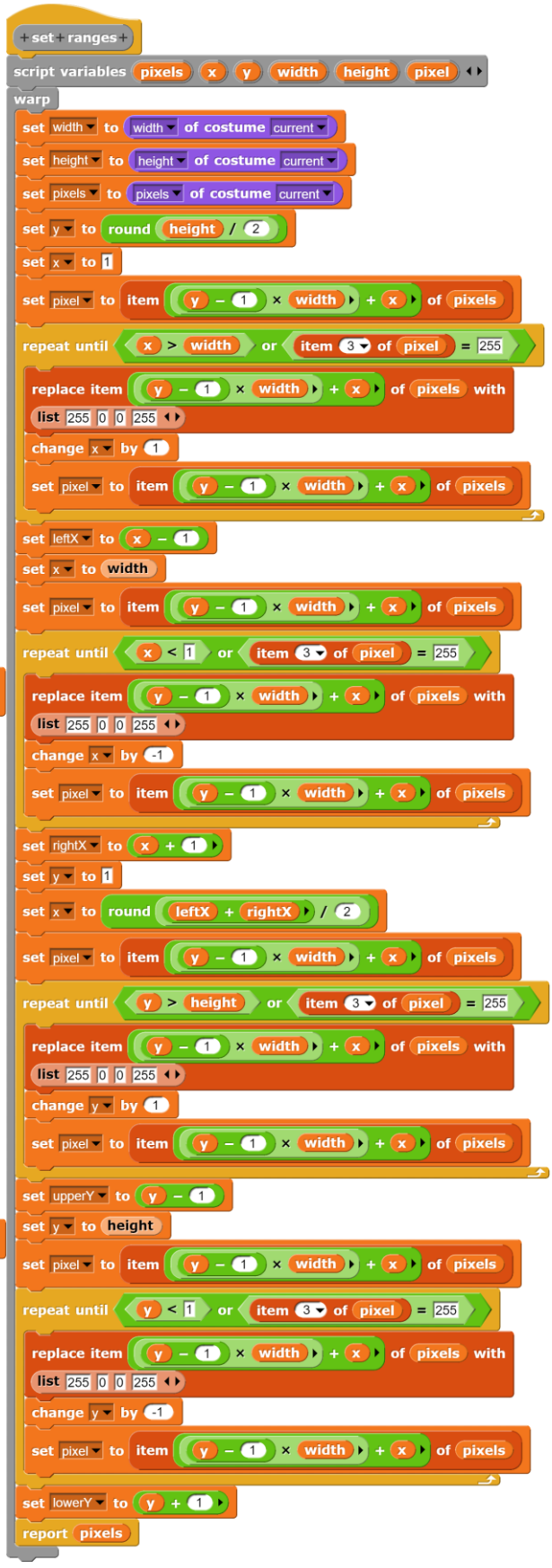
Die Grenzen des blauen Bereichs lassen sich leicht finden, wenn wir ihn von links bzw. rechts, oben und unten beginnend durchsuchen. Zur Verdeutlichung markieren wir die untersuchten Pixel rot. Das entsprechende Skript ist einfach.

Danach sind die Grenzen des blauen Bereichs als globale Variable *leftX*, *rightX*, *upperY* und *lowerY* bekannt. In diesem Bereich können wir jetzt die inneren nichtblauen Pixel zählen, hier getrennt nach oberem EURO- und unterem National-Bereich.



Wir erhalten als Gesamtskript zur Erkennung der Nationalität von EURO-Kennzeichen:





Mit diesen Ergebnissen können wir jetzt einerseits untersuchen, ob der anfängliche Lösungsansatz überhaupt sinnvoll war, und wenn, aus welchem Land das untersuchte Kennzeichen stammt.

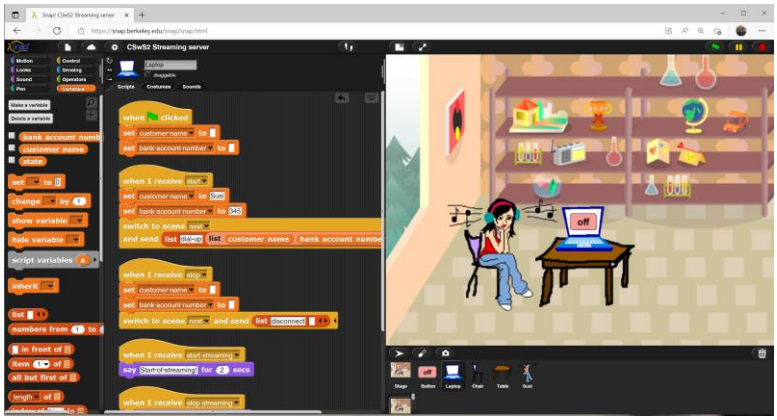
Soweit zum „technischen“ Teil. Wir können uns jetzt leicht vorstellen, dass sich auch der restliche Teil eines Kennzeichens mit etwas Aufwand bestimmen lässt. Das Ergebnis dieses Prozesses wird dann an eine andere Stelle übermittelt und dort ausgewertet. Dabei kann es sich um Mautstellen, Polizeicomputer, … handeln. Wir wollen zuerst den ziemlich „unkritischen“ Fall einer Mautstelle behandeln.

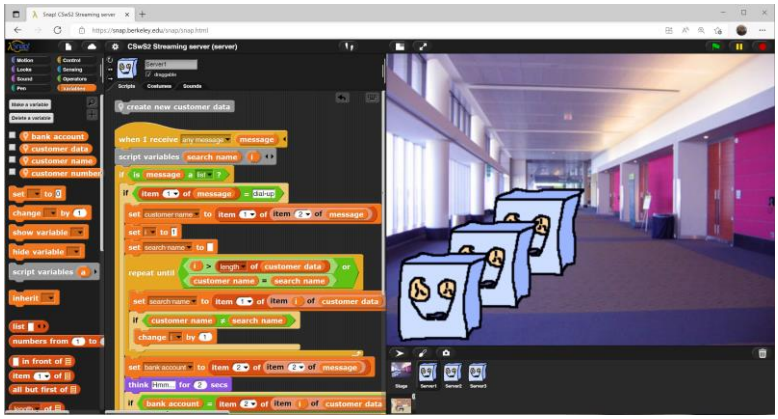
Unser Kennzeichen-Lesegerät liest also Nummernschilder und übermittelt das Ergebnis an die Zentrale – als Daten, z. B. „ABC-DE 123“. Diese Daten werden dort vom laufenden Programm ausgewertet, und zwar so, als ob es sich um Informationen handele. In diesem Fall wird z. B. angenommen, dass sich das Fahrzeug mit dem angegebenen Kennzeichen auf der Brennerautobahn befindet. Wenn die entsprechenden Voraussetzungen vorliegen, dann kann die Maut vom Konto des PKW-Halters abgebucht werden. Was passiert, wenn dieser der Abbuchung widerspricht, weil er angeblich gar nicht über den Brenner gefahren ist, sondern am Kochelsee gebadet hat? Menschliche Zeugen für beides findet man nicht, nur „der Computer“ meint, das Kennzeichen auf einem Bild identifiziert zu haben. Ist dieser Fall justiziabel? Vermutlich so nicht, weil Computer als Zeugen nicht anerkannt sind. Wahrscheinlich wird in diesem Fall auch das Originalbild, das der Computer ausgewertet hat, gespeichert worden sein, sodass menschliche Experten überprüfen können, ob sich die Maschine geirrt hat. Es lassen sich aber leicht Szenarien angeben, wo diese Überprüfung nicht erfolgt oder auch gar nicht möglich ist, z. B. weil der Betroffene nichts von seiner „Identifizierung“ erfährt. Als Beispiel mögen die Bewegungsdaten eines Handys dienen, für die es sehr unterschiedliche Interessenten gibt.

Die von der Bildauswertung übermittelten Daten haben also auch in diesem Fall wenig mit den Informationen zu tun, die aus ihnen abgeleitet werden. Wird diese Interpretation an Maschinen ausgelagert, dann kommen wir schnell zu Szenarien, die im Bereich „Informatik und Gesellschaft“ anzusiedeln sind.

Streaming

Altersstufe: *Sekundarstufe I/II*
Material: *Streaming*

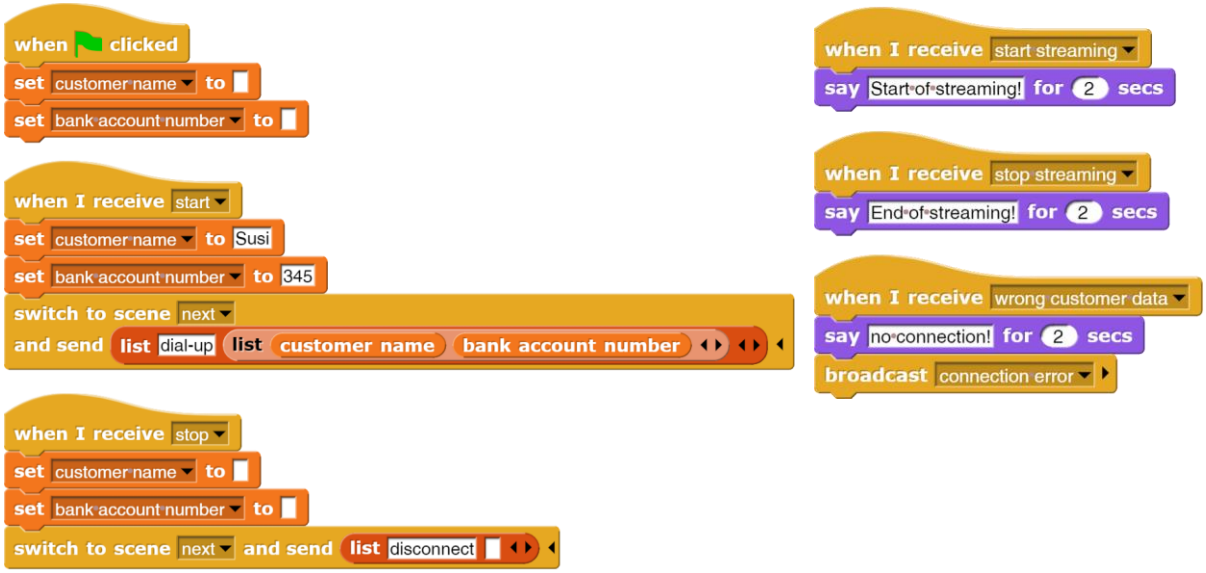




Wir benutzen ein Projekt mit zwei Szenen, bei denen eine als Zimmer dient, wo *Susi* Musik hören möchte, und die andere als Serverraum des Streaming-Dienstleisters. Auf der Server-Seite wird eine Liste von Kundendaten verwaltet, die das Einloggen ermöglicht, und eine (im Beispiel nicht realisierte) Abrechnung durchgeführt, die von der Nutzungsdauer abhängt. Ist das Nutzerkonto leer, dann wir die Verbindung abgeschaltet. Auf der Client-Seite befinden sich Susi und ihr Laptop. Dieser baut die Verbindung auf, wenn der Einschaltknopf gedrückt wird, und beendet sie beim zweiten Klicken.

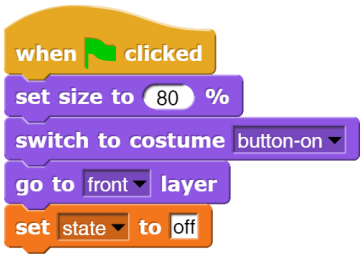
Die Schülerinnen und Schüler müssen natürlich das Abrechnungssystem auf der Serverseite erst einmal einrichten. Ihr Hauptproblem sollte aber sein, eine sichere Verbindung zwischen Server und Client herzustellen, auf der die übermittelten Daten nicht mitgelesen werden können. Da es dafür sehr unterschiedliche Lösungen gibt, handelt es sich um eine stark differenzierende Aufgabe.

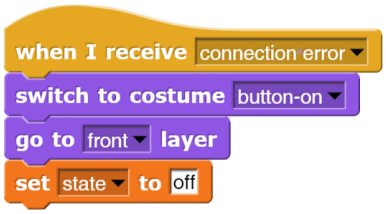
Wir realisieren die Lösung hier sehr einfach ohne Verschlüsselung über Botschaften und Szenenwechsel. Der Laptop ist z. B. für den Verbindungsaufbau zuständig. Er macht das über Szenenwechsel, denen er eine Botschaft (als Liste) anhängt.



Der Button auf dem Laptop steuert einfach das Ein- und Ausschalten.

Und Susi? Sie macht eigentlich gar nichts – außer Kostümwechseln. Sie will ja auch nichts machen, sie will chillen!





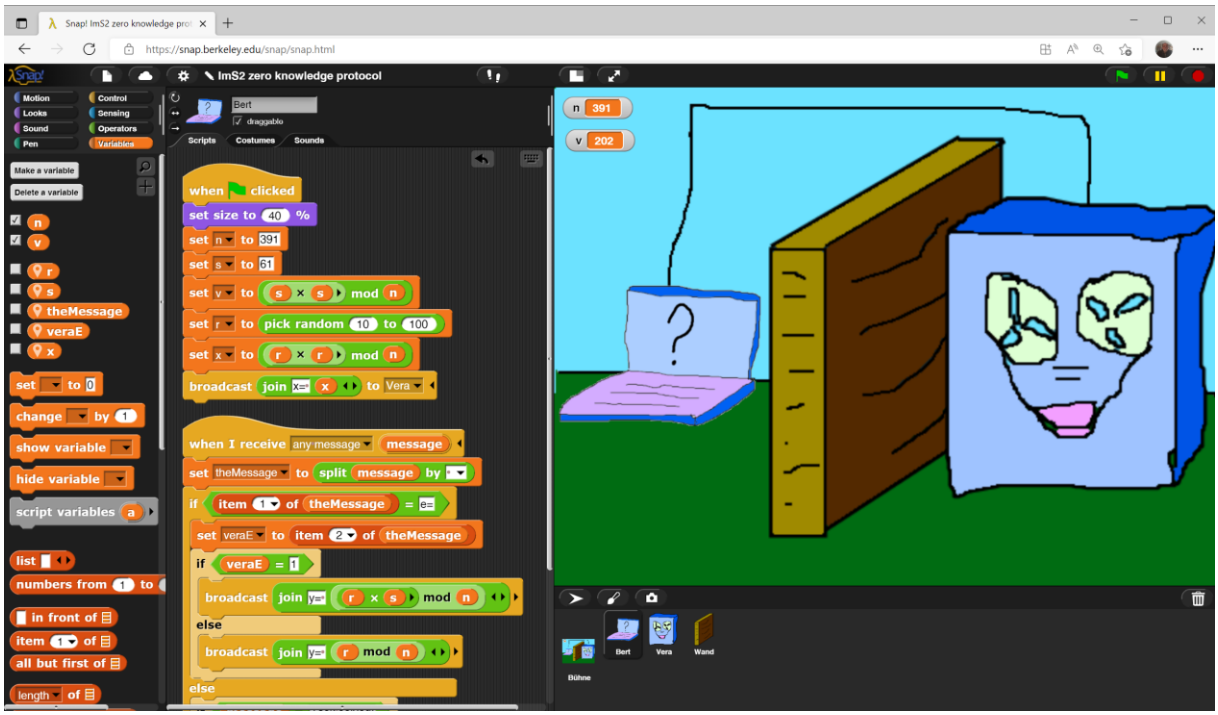
Der Streaming-Server erhält beim Szenenwechsel eine Botschaft in Form einer Liste. Der erste Eintrag enthält entweder die Nutzerdaten, die dann vom Server ausgewertet werden, oder die Bitte, das Streamen zu beenden. Und anfangs richtet er eine Liste mit Kundendaten ein.

Wie man sieht, sind die bisherigen Skripte trivial. Die Lösung lässt sich aber auf sehr unterschiedliche Weise stark ausbauen.

Die beim LogIn-Prozess übermittelten Daten sollen die Information enthalten, dass es sich beim Benutzer (in diesem Fall) um Susi handelt. Offensichtlich kann diese Information stimmen - oder auch nicht. Die Daten alleine bestimmen also nicht den Informationsgehalt, sondern der gesamte Kontext ist wichtig, z. B. sein Sicherheitsaspekt, von dem abhängt, in wieweit den Daten zu trauen ist.

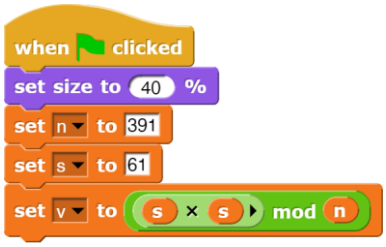
Zero Knowledge Authentifizierung

Altersstufe: *Sekundarstufe II* Material: *Zero knowledge protocol*



Die Idee des Zero-Knowledge-Protokolls[37] ist, dass ein Beweiser („Bert“) einem Verifizierer („Vera“) beweisen muss, dass er über bestimmte Informationen (den Schlüssel) verfügt, ohne dass der Beweiser diesen Schlüssel über das Netz mitteilt. Dazu stellt Vera dem Beweiser Aufgaben, deren Lösung man nur mit einer gewissen Wahrscheinlichkeit p erraten kann. Wäre $p = 0,5$ und die Anzahl der Fragen $n = 10$, dann wäre diese Fragenkette nur mit einer Wahrscheinlichkeit von $(0,5)^{10} = 0,00097..$ durch Erraten richtig zu beantworten. Wählt man n höher, dann lässt sich das antwortende System praktisch beliebig sicher authentifizieren. Wir wählen eine einfache Version des *Fiat-Shamir-Protokolls*, die wie folgt abläuft:

Voraussetzung:
Bert bestimmt eine große Zahl n als Produkt zweier großer Primzahlen: $n = p*q$. Dann wählt er eine zu n teilerfremde Zahl s und berechnet $v = s^2 \ mod \ n$. n und v veröffentlicht er. In unserem Fall geschieht das durch Wertzuweisungen an zwei globale Variable.



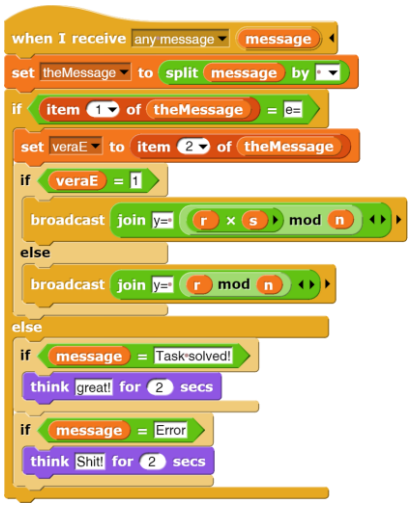
[37] https://de.wikipedia.org/wiki/Fiat-Shamir-Protokoll

Zur Authentifizierung werden dann die folgenden Schritte mehrfach durchlaufen:

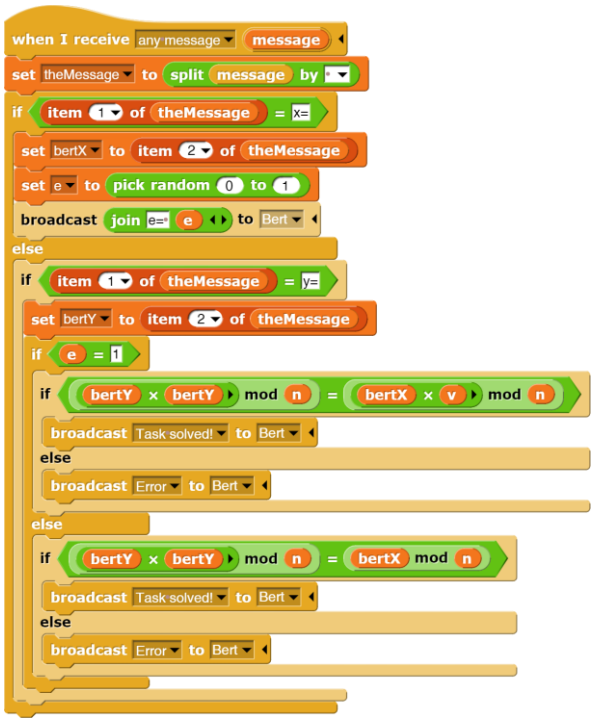
1. Bert bestimmt eine Zufallszahl *r* und sendet $x = r^2 \bmod n$ an Vera.

2. Vera merkt sich *x*, bestimmt ein zufälliges Bit *e* (*0* oder *1*) und sendet dieses an Bert.

3. Bert berechnet $y = r*s^e \bmod n$ und sendet y an Vera.



4. Vera überprüft, ob $y^2 \bmod n = x*v^e \bmod n$ ist und teilt den Erfolg bzw. Misserfolg mit.



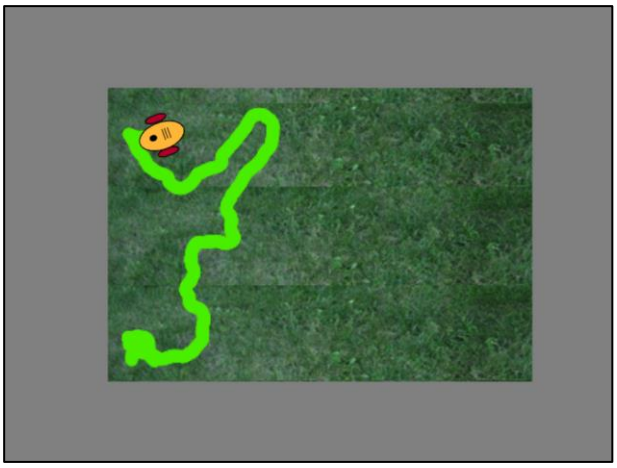
Auch in diesem Fall werden Daten zwischen den Kommunikationspartnern übertragen. Allerdings ergibt sich deren Inhalt nicht aus den übertragenen Werten, sondern aus deren Stimmigkeit innerhalb des Rahmens des Protokolls, der über den reinen Datenaustausch herausgeht. Es geht also nicht um die Daten selbst, sondern um deren Eigenschaft, „richtig“ zu sein.

4 Einfache Beispiele

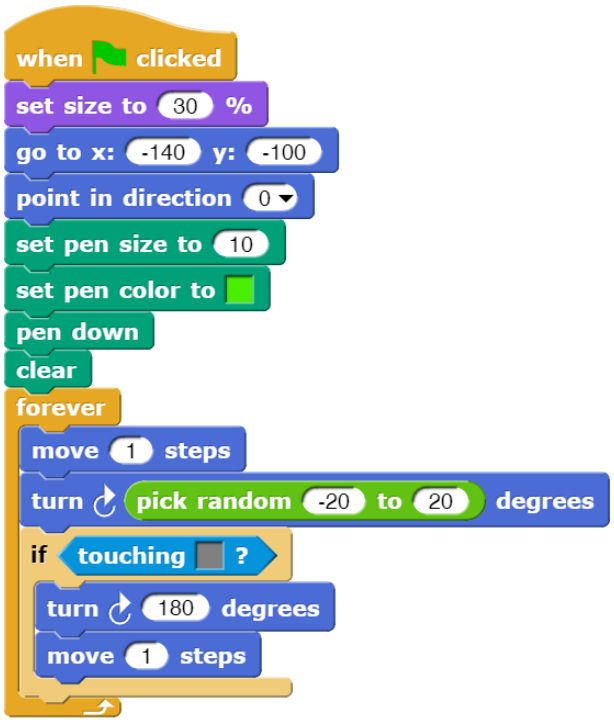
Die folgenden Beispiele demonstrieren jeweils ein paar Aspekte von *Snap!*. Sie sind schnell zu realisieren und sollten zu Modifikationen und Erweiterungen anregen. Sie zeigen vor allem, wie einfach die Visualisierung in *Snap!* ist.

4.1 Ein Rasenmähroboter

Altersstufe: *Sekundarstufe I* Material: *Lawnmower*



Wir versehen die Bühne mit dem Kostüm eines Rasens mit grauem Rand, verwandeln sie also in einen modernen Lavastein-Garten. Für das Sprite zeichnen wir einen Rasenmäh-Roboter als neues Kostüm. Der Roboter soll den Rasen mähen, und das kann er auf sehr unterschiedliche Art tun. Wir realisieren eine einfache, die nur zufallsgesteuert abläuft. Der Roboter überschreibt dabei den Rasen-Hintergrund mit der hellgrüner Farbe eines frisch gemähten Rasens.

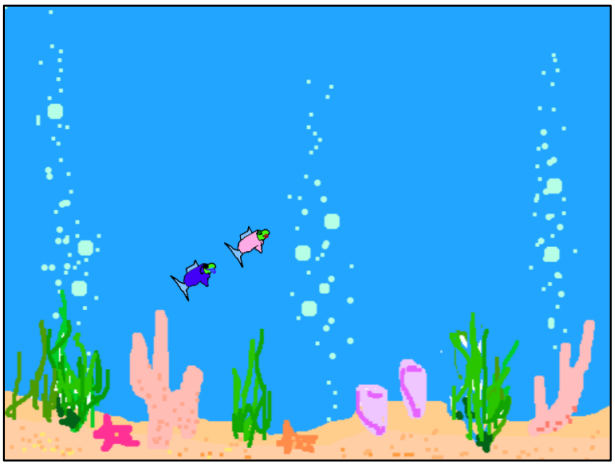


Die Aufgabe ist trotzdem nicht trivial. Wird der Rasen z. B. im Inneren immer vollständig gemäht? Was ist mit dem nicht gemähten Streifen am Rand? Gibt es geeignetere Roboterbewegungen für das Mähen? Welche schafft es am schnellsten? Kann man eine Zeitmessung installieren? Wo ist die Ladestation des Roboters und wann soll er die anfahren? Was geschieht mit den Pflanzen im Rasen, z. B. den Frühlingstulpen? Wächst der Rasen nach?

Es kann beliebig kompliziert werden – und zu allen Fragen gibt es ganz unterschiedliche Lösungsmöglichkeiten.

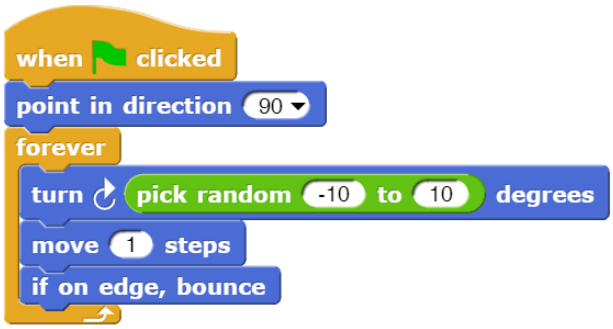
4.2 Im Aquarium

Altersstufe: *Sekundarstufe II* Material: *Aquarium*

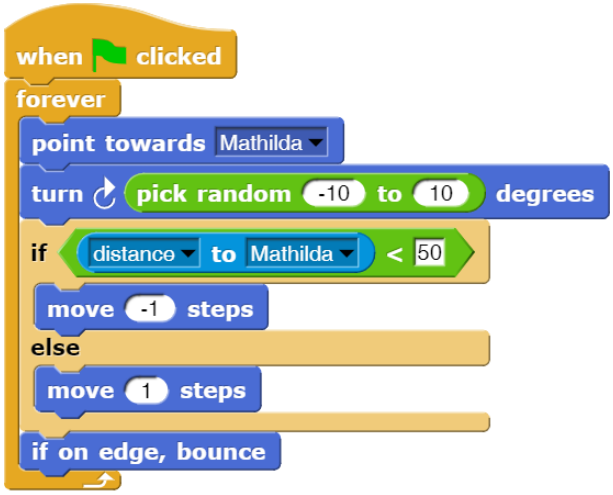


Wir suchen uns ein schönes Hintergrundbild für die Bühne (oder zeichnen uns eins) und importieren es als Kostüm der *Stage*. Danach erzeugen wir zwei Sprites mithilfe des -Knopfes im Sprite-Coral und nennen sie *Mathilda* und *Joe* – wie sonst. Für diese zeichnen wir jeweils ein Fisch-Kostüm.

Mathilda ist wenig an anderen Fischen interessiert und schwimmt unabhängig von diesen im Aquarium umher. Trifft sie auf eine Wand, dann kehrt sie um.



Joe ist eher an Mathilda als am restlichen Aquarium interessiert. Er schaut dauernd in ihre Richtung und schwimmt auf sie zu. Kommt er ihr zu nahe, dann geht er aber vorsichtig auf Distanz. Er hat da so seine Erfahrungen.

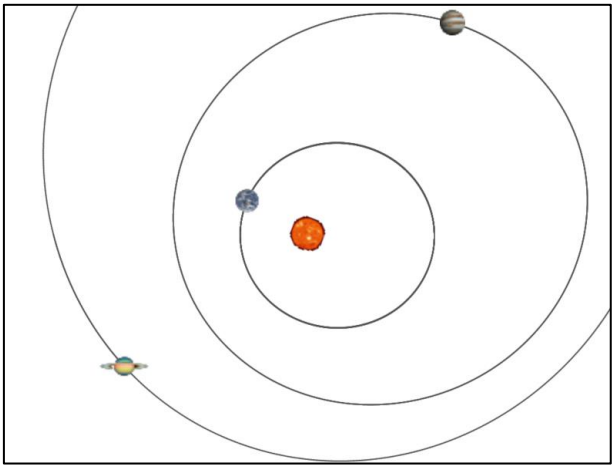


Wir können sehr einfach weitere Fische erzeugen, die insgesamt eine Kette bilden, wie man sie manchmal in großen Seeaquarien beobachten kann. Auch die Einführung eines Hais ist einfach. Der schwimmt auf andere Fische zu, aber wenn er ihnen zu nahekommt, dann hauen die schnell ab.

Anspruchsvoller ist eine echte Schwarmbildung, bei der viele Fische eine gemeinsame Struktur aufbauen. Die Strategien dafür sind im Netz zu finden[38] und dann auch gut zu realisieren.

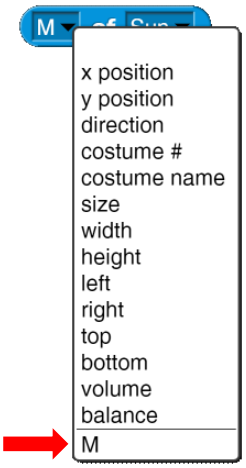
[38] z. B. in https://de.wikipedia.org/wiki/Schwarmverhalten

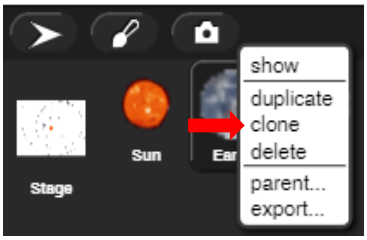
4.3 Das Sonnensystem[39]



Altersstufe: *Sekundarstufe II* Material: *Solar system*

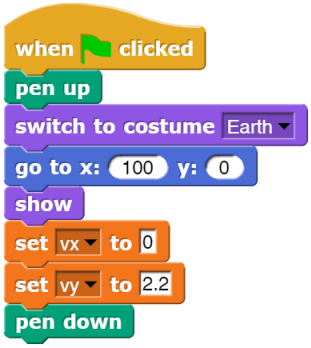
Befindet sich eine Sonne der Masse *M* im Ursprung des Koordinatensystems, dann erhält man die Gravitationskraft auf einen Planeten der Masse *m* zu $\vec{F} = -G \cdot \frac{m \cdot M}{r^3} \cdot \vec{r}$, also $\vec{a} = -G \cdot \frac{M}{r^3} \cdot \vec{r}$. Den Wert der lokalen Sonnen-Variablen *M* erhalten andere Sprites mithilfe des … *of* …-Blocks (s. Bild).





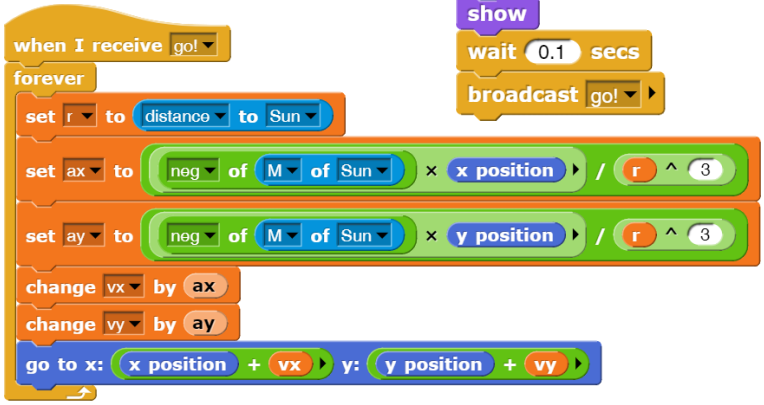
Wir holen uns ein Bild der Sonne und einige Planetenbilder aus dem Netz und verkleinern sie stark. Dann laden wir sie als Kostüme in ein Planeten-Prototyp-Sprite namens *Earth*. Ein zweites Sprite namens *Sun* räumt den Bildschirm auf und startet die Simulation. Andere Planeten werden durch Klonen der Erde erzeugt

Unsere Erde verfügt über einen Satz lokaler Variablen, die ihren Zustand beschreiben. Dazu gehören die Geschwindigkeitskomponenten *vx* und *vy*, die Beschleunigungskomponenten *ax* und *ay* sowie der Abstand von der Sonne *r*. Die Geschwindigkeitswerte werden beim Start der Simulation durch Klicken der grünen Flagge jeweils geeignet gesetzt.



Die Sonne löscht den Bildschirm, setzt ihre Masse und zeigt sich in der Mitte. Danach wartet sie einen Augenblick, damit die Planeten mit der Selbstinitialisierung genug Zeit haben, und startet die Simulation durch Aussenden der Botschaft „*go!*“. Danach stellt die Sonne ihre Tätigkeit ein.

Die Planeten reagieren auf die Botschaft, indem sie ihren aktuellen Abstand zur Sonne messen. Danach berechnen Sie die Beschleunigungskomponenten *ax* und *ay*. Diese ändern die entsprechenden Geschwindigkeitskomponenten *vx* und *vy*, und daraus lässt sich die neue Planetenposition berechnen. Diese Vorgänge werden dauernd wiederholt und ergeben die Planetenbahnen. Alle Werte wurden so gewählt, dass die Bahnkurven wenigstens teilweise auf den Bildschirm passen.



[39] In einer ziemlich vereinfachten Version: die Sonne steht wie angenagelt in der Mitte und die Planeten beeinflussen sich nicht.

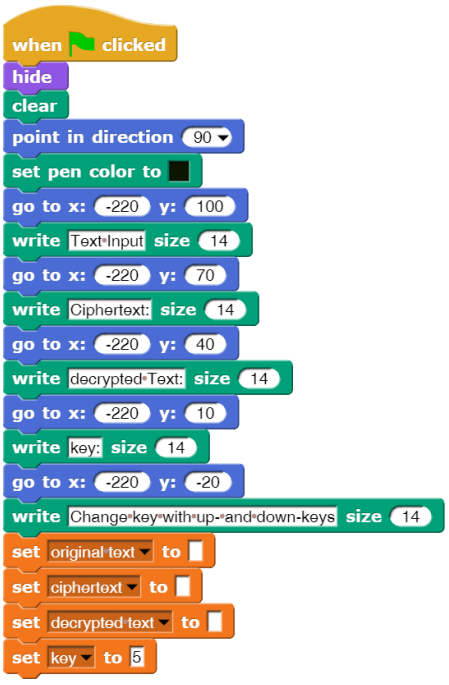
4.4 Caesar-Verschlüsselung

Altersstufe: *Sekundarstufe I/II*
Material: *Caesar encryption*

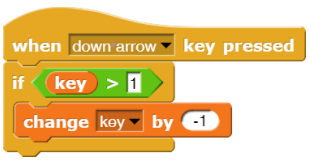
Wir wollen einfache Zeichenketten nach dem Caesar-Verfahren ver- und entschlüsseln. Da es sich hierbei um knallharte Informatik handelt, benötigen wir eine sehr seriöse, etwas langweilige Oberfläche, auf der sich ein paar Buttons befinden. Diese importieren wir aus der *Costumes*-Bibliothek mithilfe des Datei-Menüs.[40] Das *Button*-Bild exportieren wir in eine Datei. Mithilfe eines Grafikprogramms ziehen wir es etwas in die Länge und beschriften es unterschiedlich. Die entstandenen Kostüme importieren wir wieder.

Wir erzeugen drei neue leere Blöcke namens *text input*, *encryption* und *decryption* und sorgen dafür, dass unsere Buttons beim Anklicken mit dem Aufrufen eines dieser Blöcke reagieren. Dazu kopieren wir den Button zweimal mithilfe des Kontextmenüs im Sprite-Bereich und ändern die Kostüme und aufgerufenen Blöcke entsprechend. Wir ziehen die Buttons an die richtige Stelle, ändern ihre Namen z. B. in *bTextinput* und entfernen den Haken vor dem Kästchen *draggable*. Damit sitzt der Button fest.

Dann erzeugen wir vier globale Variable namens *original text*, *ciphertext*, *decrypted text* und *key* sowie ein neues Sprite namens *control*, das für eine sehr seriöse Oberfläche sorgt. Dazu schreibt es ein paar Texte auf die Bühne. Die vier Variablen lassen wir am Bildschirm mit Monitoren anzeigen (Häkchen vor den Variablennamen setzen) und wechseln mithilfe der Kontextmenüs im Anzeigebereich zur großen Darstellung. Dann ziehen wir diese an geeignete Stellen hinter den Texten.



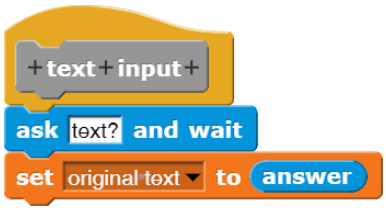
Zuletzt ermöglichen wir die Änderung des Schlüssels mithilfe von Tastendrücken.



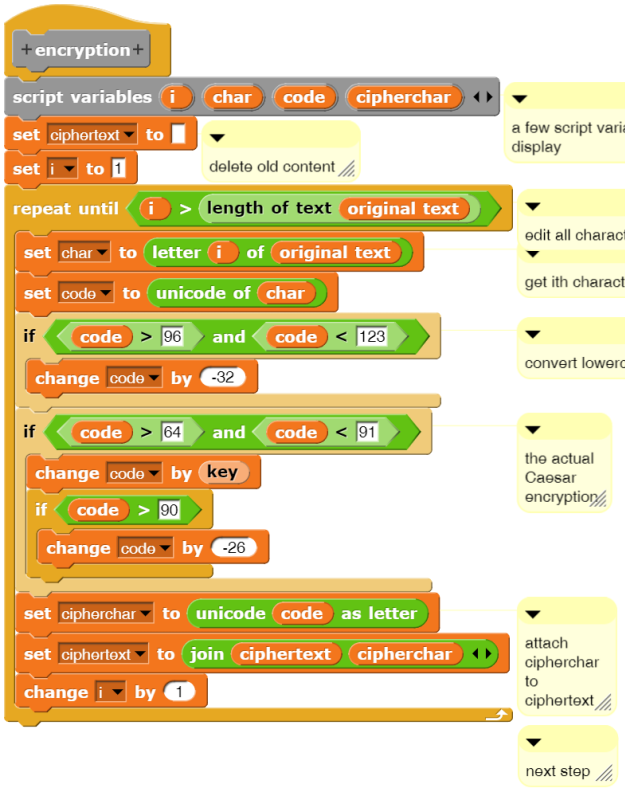
[40] Wie man sieht, befinden sich in der Bibliothek auch weitaus „interessantere“ Kostüme!

Damit ist unsere „Benutzeroberfläche“ fertig.

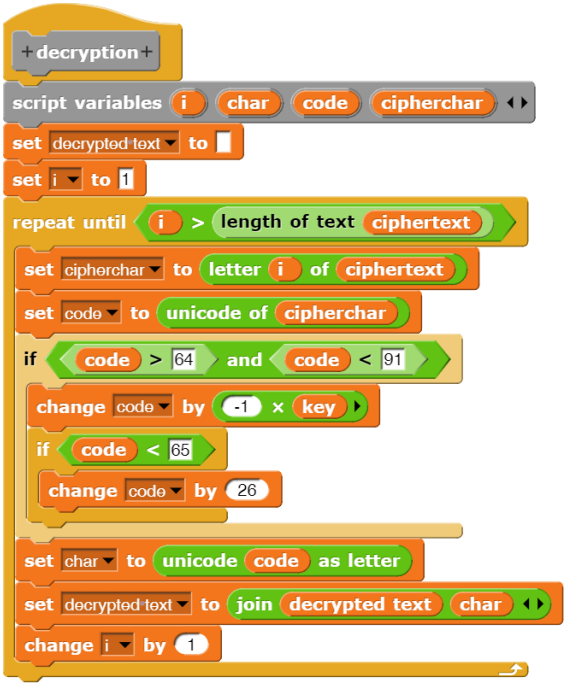
Wir kommen jetzt zur eigentlichen Funktionalität, die unabhängig voneinander entwickelt werden kann. Die Texteingabe ist einfach: wir fragen einfach nach dem Originaltext. Die Ausgabe kann man dabei auch schöner gestalten.



Die Caesarverschlüsselung besteht daraus, alle Zeichen im Code (hier: im Unicode) um die Schlüssellänge zu verschieben. Die letzten Zeichen werden dabei zyklisch nach vorne verschoben. Im nebenstehenden Skript erfolgt das sehr weitschweifig, aber – hoffentlich – lesbar. Zu beachten ist dabei, dass der grüne *length of text <string>*-Block aus der *Operators*-Palette mit Zeichenketten arbeitet, die braune *<length> of <list>*-Version aus der *Variables*-Palette dagegen mit Listen.

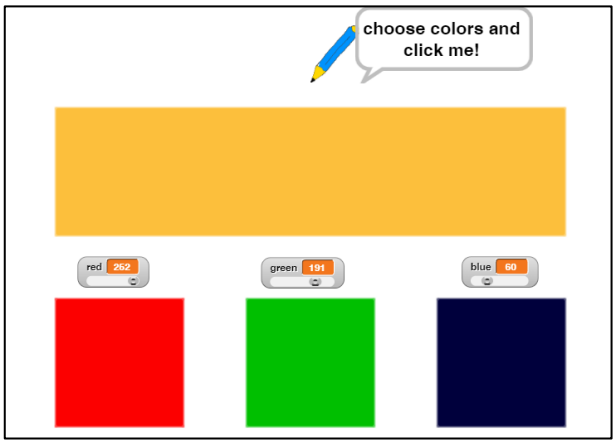


Das Entschlüsseln erfolgt direkt invers zum Verschlüsseln.



4.5 Farbmischer

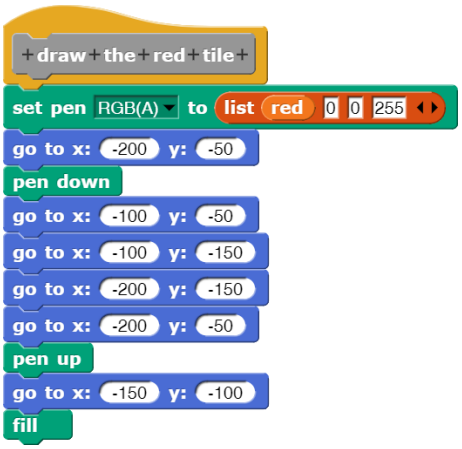
Altersstufe: *Sekundarstufe I/II* Material: *Color mixer*



Wir wollen in der üblichen Art aus den Grundfarben Rot, Grün und Blau Mischfarben erzeugen und darstellen. Dazu muss man wissen, dass Farben im RGB-Modell durch 4-elementige Listen dargestellt werden, die die drei Farbwerte und zusätzlich die „Transparenz“ der Mischfarben, also ihre Deckkraft, enthalten. Alle vier Werte stammen aus dem Zahlenbereich 0 bis 255, können also jeweils durch ein Byte gespeichert werden. Soll der Stift in vollem Rot zeichnen, dann wird das durch den nebenstehenden Block erreicht.

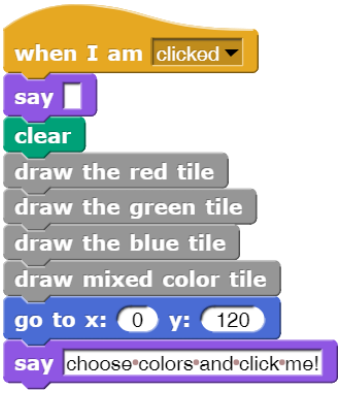
Wir erzeugen also drei Variable mit den Bezeichnern *red*, *green* und *blue*, lassen sie auf der Bühne darstellen (Häkchen vor dem Variablennamen setzen) und platzieren sie an geeigneter Stelle. Danach wählen wir aus deren Kontextmenüs den Punkt *slider* aus, damit ein Schieberegler unter dem Variablenwert erscheint, und setzen danach den Wert für *slider max* auf *255*. Damit können wir die gewünschten Farbwerte einfach durch Verschieben des Reglers auswählen.

Nach diesen Vorbereitungen können wir unter die Rot-Variablen ein entsprechend gefärbtes Quadrat zeichnen lassen, und mit leicht veränderten Koordinaten auch das grüne und das blaue. Und natürlich gehört als Krönung des Projekts ein größeres Rechteck mit der Mischfarbe über alles.



Gestartet wird der Zeichenvorgang durch Anklicken eines Zeichenstifts, dem wir dafür auch ein geeignetes Kostüm zuordnen.

Erarbeitet man zusammen das Grundgerüst dieses Projekts, das ein rotes Rechteck und die Reaktion auf das *OnClick*-Ereignis des Stifts enthält, dann können die fehlenden drei Farbflächen in direkter Analogie von den Lernenden selbst erzeugt werden. Und es gibt natürlich Verbesserungsmöglichkeiten:



- Die Farben sollten sich doch sofort ändern, wenn ein Farbwert geändert wird, und nicht erst beim Klicken auf den Stift.
- Die Hilfe ist auch sehr sparsam gehalten. Das kann man besser machen!
- Aus dem Physikunterricht kennt man Farbkreise, ggf. mit unterschiedlicher Transparenz, die die Mischfarben anzeigen. Geht das auch in *Snap!*?
- Gibt es andere Farbmodelle als das RGB-Modell? Welche? Wie funktionieren die?

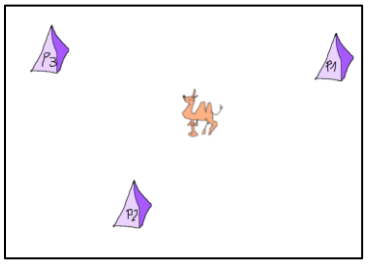
4.6 Aufgaben

1. a: Informieren Sie sich über die **XOR-Verschlüsselung**. Implementieren Sie das Verfahren.

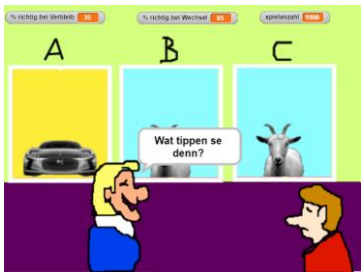
   b: Informieren Sie sich über **Versetzungsverfahren** bei der Verschlüsselung. Implementieren Sie das Verfahren.

   c: Informieren Sie sich über die **Kryptoanalyse**. Implementieren Sie eine Häufigkeitsanalyse.

2. Beim **Kamelproblem** gerät das Tier in eine furchtbare Situation zwischen drei Pyramiden. Es bewegt sich zielstrebig auf eine zufällig ausgewählte Pyramide zu. Hat es genau die Hälfte der Entfernung zur Pyramide zurückgelegt, kommt ein gehässiger Wüstengeist und wirbelt das arme Geschöpf herum, sodass es nicht mehr weiß, welche Pyramide es ansteuerte. Die Bewegung hinterlässt natürlich einen Abdruck auf dem Bildschirm, und die Prozedur beginnt von neuem.

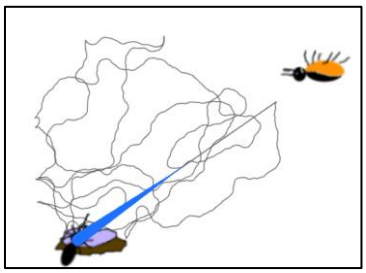


3. Das **Ziegenproblem** taucht immer mal wieder in den Medien auf. Es geht darum: Bei einem Gewinnspiel gibt es drei Türen, hinter denen sich bei zweien eine Ziege befindet, hinter der dritten ist der Hauptgewinn. Der Spielleiter, der die Positionen kennt, bittet den Spieler, eine Tür zu raten. Danach öffnet er eine der verbleibenden Türen, hinter der sich eine Ziege befindet, und bietet dem Spieler an, sich umzuentscheiden – oder nicht. Die Frage ist: sollte der das tun? Realisieren Sie das Spiel und entscheiden Sie die Frage empirisch.



4. a: **Wüstenameisen** leben alleine in der Wüste. Verlassen sie den Bau, dann suchen sie in der Umgebung nach etwas Fressbarem. Haben sie das gefunden, dann laufen sie direkt zum Bau zurück. Offensichtlich merken sie sich, welche Bewegungen sie insgesamt ausgeführt haben. Aus diesen „berechnen“ sie den direkten Weg zurück. Realisieren sie das Verfahren.

   b: Auf ihrem Weg zum Bau sollen die Ameisen eine **Pheromonspur**, die langsam verdunstet. Auf dieser finden sie zur Beute zurück, holen ein weiteres Stückchen und laufen zurück zum Bau, wobei sie eine neue Pheromonspur legen. Haben sie nichts mehr gefunden, dann hinterlassen sie keine neue Spur.



5 Simulation eines Federpendels

Altersstufe: *Sekundarstufe II* Material: *Spring pendulum*

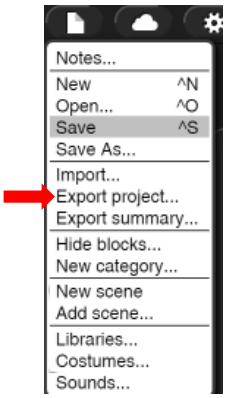
Neben der weitgehenden Syntaxfreiheit sind die exzellenten Visualisierungsmöglichkeiten und das gutmütige Verhalten von *Snap!* bei Fehlern ein Anreiz für die Lernenden, experimentell vorzugehen und so eigene Ideen zu erproben. Experimentelles Vorgehen öffnet so gerade am Anfang Möglichkeiten zum selbstständigen 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 Probleme, 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!

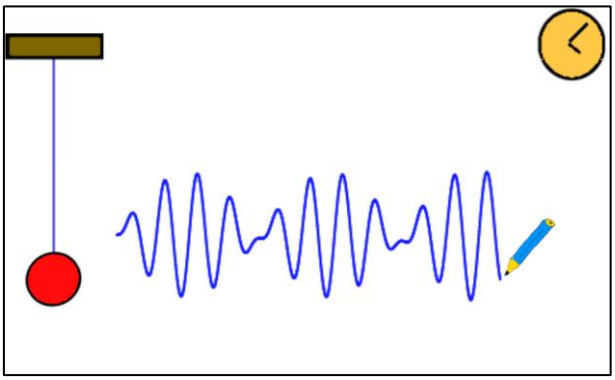
Organisation von Zusammenarbeit

Wenn Gruppen weitgehend unabhängig voneinander arbeiten, muss einerseits klar sein, in welchem Rahmen gearbeitet wird, andererseits, wie die Ergebnisse später zusammengetragen werden können.

Um einen Rahmen zu erzeugen, kann man leere Blöcke mit den richtigen Namen als „Dummies“ erzeugen. Diese können in Skripten benutzt werden, noch ohne die gesuchte Funktionalität. Die benötigten Objekte können ebenfalls erzeugt und mit rudimentärem Verhalten ausgestattet werden, z. B. als Reaktion auf Ereignisse: Sie können z. B. eine Sprechblase ausgeben mit einem erläuternden Text: *„Jetzt sollte eigentlich das und das passieren!“* Dieser Programmrahmen kann insgesamt oder in Teilen exportiert und importiert werden:



- Das Projekt kann mit allen seinen Teilen mithilfe des Datei-Menüs exportiert werden. Daraufhin erscheint es unten im *Snap!*-Fenster. Ein Klick auf den Pfeil rechts davon führt zum Download-Ordner, wo es gespeichert wurde. Von dort kann es verschickt, importiert (s. Bild) oder in irgendein *Snap!*-Fenster gezogen und so wieder geöffnet werden.
- Gibt es globale Methoden (Blocks *„for all sprites“*) im Projekt, dann erscheint im selben Menü ein weiterer Punkt *„Export blocks…“*. Wird der ausgewählt, dann können im erscheinenden Fenster die zu exportierenden Blöcke ausgewählt werden. Diese können wie Projekte in geöffnete *Snap!*-Fenster gezogen, importiert oder verschickt werden.
- Sprites können mit ihren lokalen Methoden als Ganzes exportiert werden, indem in ihrem Kontextmenü im Sprite-Bereich der Punkt *„export…“* gewählt wird. Der Re-Import erfolgt wie oben geschildert.
- Innerhalb eines Projektes können Skripte von einem Objekt zum nächsten transferiert werden, indem sie vom Sprite, in dem sie sich auf der Skriptebene befinden, auf das Sprite im Sprite-Bereich, das mit dem Skript versorgt werden soll, gezogen werden. Der Adressat wird beim „Draufziehen“ etwas hervorgehoben, wenn er gemerkt hat, dass er gemeint ist.



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

Wir identifizieren

- einen *Erreger*, die dunkle Platte oben-links, der periodisch vertikal schwingt. Seine Frequenz ω ist eine Instanzvariable und kann in der Variablenanzeige geändert werden.
- eine *Kugel*, die relativ dumm am Faden hängt, aber immerhin 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.

Die Uhr

Wir erzeugen ein neues Sprite und zeichnen eine einfache Uhr als deren Kostüm. Beim Anklicken der grünen Flagge wählen wir dieses Kostüm für die Uhr und schicken sie in die Ecke rechts-oben. Nachdem die Uhr mithilfe der *start*-Nachricht (grüne Flagge angeklickt) gestartet wurde, setzt sie den in *Snap!* eingebauten *Timer* zurück und merkt sich fortlaufend die aktuelle Zeit in der Variablen *t*, die wir auch anzeigen.[41] Da die Zeit *t* logisch zur Uhr gehört, vereinbaren wir sie als lokale Variable. Der Zugriff auf lokale Variable erfolgt von anderen Objekten aus über den *<attribute> of <object>* - Block der *Sensing*-Palette. Wir exportieren das Uhr-Sprite wie angegeben in die Datei *Clock.xml*.

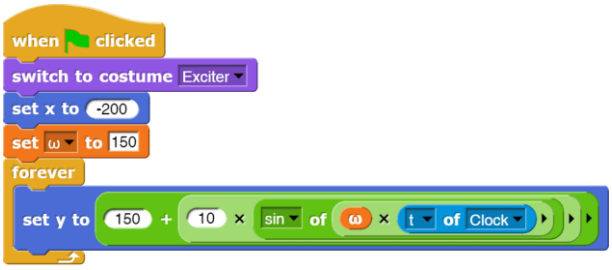
Erweiterung: Lassen Sie die Uhrzeit vom Sprite anzeigen, indem die Zeiger richtig bewegt werden.

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: -200) sowie eine Ruhelage in *y*-Richtung (hier: 150). 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

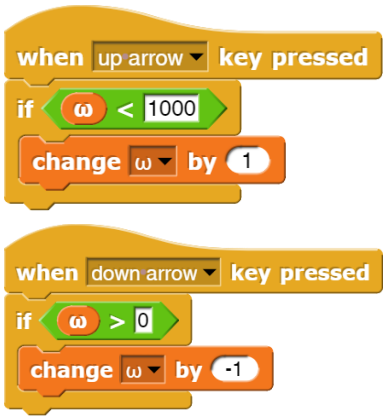
$$y = 150 + 10 * sin(\omega t).$$

Diese Angaben lassen sich direkt in ein Skript übersetzen.



[41] Natürlich hätten wir stattdessen auch direkt auf die Timer-Zeit zugreifen können. Aber ich möchte den Zugriff auf lokale Variable von anderen Objekten aus zeigen.

Das Skript beginnt seine Arbeit, wenn die *start*-Botschaft gesendet wird. Da die Skripte der anderen Teile zum gleichen Zeitpunkt gestartet werden müssen, bietet sich diese Möglichkeit an.



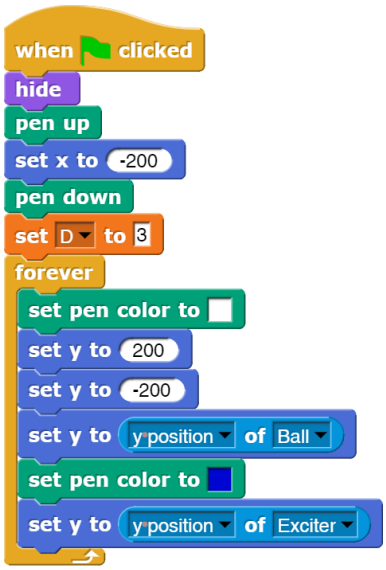
Interessanter sind die benutzten Variablen. Die Zeit wird von der Uhr importiert. Die Frequenz wird in keinem anderen Skript benötigt und sollte daher lokal vereinbart werden. Man kann sie mithilfe der Pfeiltasten ändern.

Wir exportieren das Sprite wie beschrieben als *Exciter.xml*.

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.

Der Faden

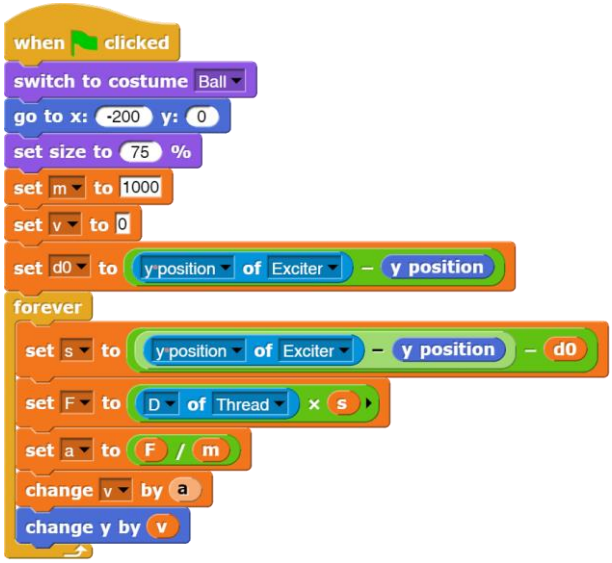
Der Faden ersetzt die Feder. Er verfügt nur über eine einzige Eigenschaft, die Federkonstante *D*. Diese wird einmal auf einen festen Wert gesetzt, danach wird eine helle senkrechte Linie am Ort des Fadens gezeichnet, die seine alte Darstellung löscht (das geht natürlich auch eleganter). Danach wird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



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

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.

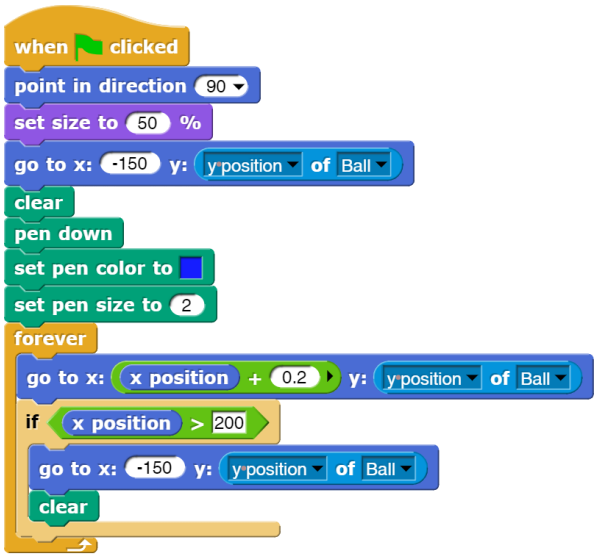


Wir exportieren die Kugel als *Ball.xml*.

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.

Der Stift

Der Stift verfügt über keine 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 die Klasse als *Pen.xml*.

Erweiterung: Führen Sie eine Möglichkeit ein, dass der Stift mit unterschiedlichen Geschwindigkeiten laufen kann.

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. Trotzdem treten sie in der Simulation auf. 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. Als Werkzeug dafür nutzen wir statt der Mathematik einen Algorithmus, der das Systemverhalten über eine Folge von kleinen zeitlichen Änderungen verfolgt. Statt also „mathematisch“ zu integrieren, iterieren wir „informatisch“. Außer in den einfachen Fällen macht ein Tool zur Integration eines Differentialgleichungssystems auch nichts anderes.

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.

6 Fehlersuche in *Snap!*

Altersstufe: *Sekundarstufe II* Material: *Towers of Hanoi*

Snap! visualisiert den Programmablauf, ohne dass besondere Aktivitäten der Lernenden erforderlich sind. Schon dadurch werden viele Fehler „sichtbar“, für deren Auffinden sonst die mühsame Analyse von Code erforderlich wäre. Bewegt sich ein Körper z. B. in die falsche Richtung, dann ist ziemlich klar, wonach gesucht werden muss.

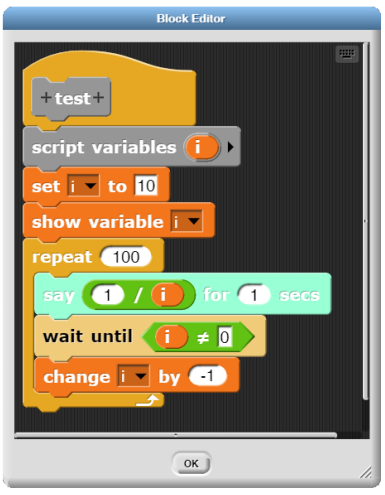
Da globale und lokale Variable durch Setzen der Häkchen vor den Variablenname in einem *Monitor* auf der Bühne angezeigt werden können, ist deren Veränderung ebenfalls direkt beobachtbar. Skriptvariable können genauso angezeigt werden, wenn im Skript die Blöcke *show variable <name>* bzw. *hide variable <name>* eingebaut werden. Ein wesentlicher Aspekt bei der Fehlersuche ist das „Einfrieren“ der Variablenbelegungen bei einem Programmstopp: unterbricht oder beendet man das Programm, dann bleiben die momentanen Werte der Variablen erhalten und können inspiziert werden.

Kontrollausgaben während des Programmlaufs können leicht mit den Blöcken der *Looks*-Palette erreicht werden: *say <irgendwas> for <n> secs* und seine Verwandten gestatten auch die Ausgabe komplexerer Ausdrücke, sodass die am Bildschirm verfolgt werden können. Die *wait <n> secs* und *wait until <bedingung>-Blöcke* ermöglichen Pausen im Programmablauf an bestimmten Stellen und/oder beim Eintreten bestimmter Bedingungen.

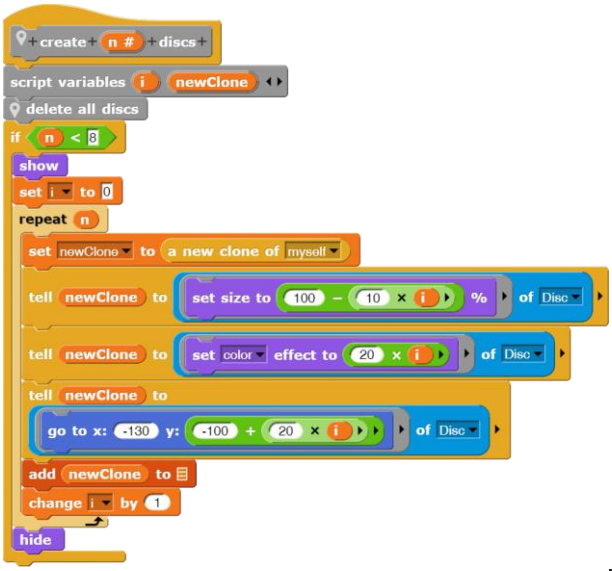
Soll der Ablauf des gesamten Programms schrittweise verfolgt werden, dann muss das *Visual Stepping* oben neben dem Ausgabefenster eingeschaltet werden.

Danach erscheinen die Fußstapfen hellgrün und neben ihnen erscheint ein Regler, der die Schrittgeschwindigkeit bestimmt. Zwischen grüner Flagge und rotem Stopp-Button erscheint ein Knopf zum Unterbrechen bzw. Starten des Steppings. Steht der Geschwindigkeitsregler ganz links, dann kann das Programm in Einzelschritten durchlaufen werden. Der aktuell ausgeführte Block erscheint hellgrün.

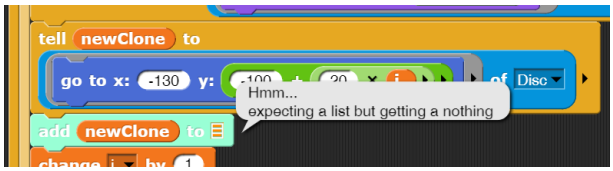
Soll der Programmablauf auch innerhalb der eigenen Blöcken verfolgt werden, dann müssen diese vor dem Start des Programms im Editor geöffnet werden. Die Blöcke können dabei auch geschachtelt sein.



Wir wollen die Abläufe anhand eines kleines Beispiels verfolgen. Warum auch immer – es soll das Problem der „Türme von Hanoi“ bearbeitet werden. Dafür zeichnen wir eine Scheibe und weisen dieses Kostüm einem Sprite *Disk* zu. Weitere Scheiben sollen durch Klonen erzeugt werden. Dafür haben wir eine Methode *create <n> discs* geschrieben – aber sie funktioniert nicht. Schade!

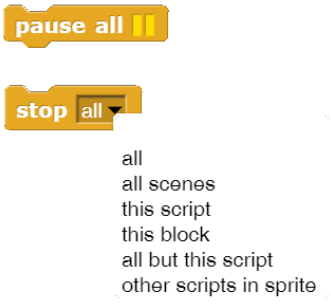


Zum Lokalisieren des Fehlers öffnen wir die Methode im Editor, klicken auf den *Visible Stepping*-Button, stellen die gewünschte Geschwindigkeit ein und klicken danach noch einmal auf den neuen Block. Im Editor können wir verfolgen, wie die Befehle aufgerufen werden – und wo es schiefgeht.



Da fehlt doch was! Wir ergänzen die fehlende Listenvariable *stackA* im Block, und dieser Teil zumindest läuft dann prima.

Weitere Blöcke, die hilfreich bei der Fehlersuche sein können, finden sich in den Bibliotheken. Sie werden durch eigene Hilfeseiten beschrieben, die über ihre Kontextmenüs aufgerufen werden.



Die – für mich – wesentlichste Möglichkeit zur Fehlersuche ist es aber, Blöcke aus den Skripten herauszunehmen und daneben „einfach liegenzulassen“. Funktioniert ein Skript danach, können die Blöcke nacheinander wieder eingefügt werden. Meist lässt sich der Fehler dadurch schnell eingrenzen.

7 Listen und verwandte Strukturen

Snap! kennt neben atomaren Datentypen wie *Zahlen*, *Wahrheitsswerten* und *Zeichen* die strukturierten Typen *String* und *List*. Die Zeichenketten folgen später, weil sie sehr viele Anwendungen ermöglichen. In diesem Abschnitt werden erst einmal Listen thematisiert, die man praktisch immer braucht. Aus ihnen lassen sich alle höheren Strukturen leicht aufbauen. Die Verwendung von Listen wird zuerst an einem einfachen Fall – dem Sortieren – gezeigt, danach folgen komplexere Anwendungen.

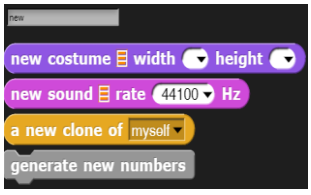
Bei Listen handelt es sich um sogenannte *Referenzen*, also Adressen, die auf einen bestimmten Speicherbereich „zeigen“, in dem sich die eigentlichen Daten befinden. Wenn man das nicht beachtet, können ärgerliche Fehler auftreten. Zum Beispiel können mehrere Listenvariablen auf den gleichen Datenbereich zeigen. Ändert man z. B. in einem Block diese Daten durch Zugriffe über eine Listenvariable, dann wirken sich die Änderungen auch auf die anderen Variablen aus, solange die auf die gleichen Daten verweisen. Solche Fehler kann man z. B. vermeiden, indem Kopien der Listen angelegt werden, mit denen man dann arbeitet, ohne die restlichen Operationen zu stören.

7.1 Sortieren mit Listen – durch Auswahl

Altersstufe: *Sekundarstufe I/II* Material: *Selection sort*

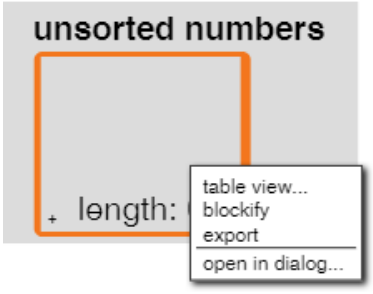
Das Beispiel ist extrem einfach gehalten: es benutzt nur globale Variable und Blöcke ohne Parameter, also Makros, die zur Zusammenfassung einer Befehlsfolge unter einem neuen Namen dienen. Da es zusätzlich die Visualisierungsmöglichkeiten von *Snap!* ausnutzt, ist es als Einführungsbeispiel gut geeignet.

Wir beginnen mit einem leeren *Snap!*-Projekt. Wenn wir etwas sortieren wollen, dann müssen die zu sortierenden Elemente irgendwo gespeichert werden. Dafür gibt es Variable, die man sich als „Boxen“ vorstellen kann, die beliebige Inhalte aufnehmen können. Für die Speicherung mehrerer Elemente gibt es Listen, eine Art „Boxenreihe“. Die Blöcke zur Bearbeitung von Variablen und Listen finden wir in der *Variables*-Palette in Braun.



Nebenbei: Das Vergrößerungsglas zum Suchen oben-rechts in den Paletten zeigt uns Kandidaten für Blöcke an, die dem Suchmuster entsprechen. Darunter finden wir auch selbst geschriebene Blöcke und einige, die sich gar nicht in den Paletten befinden.

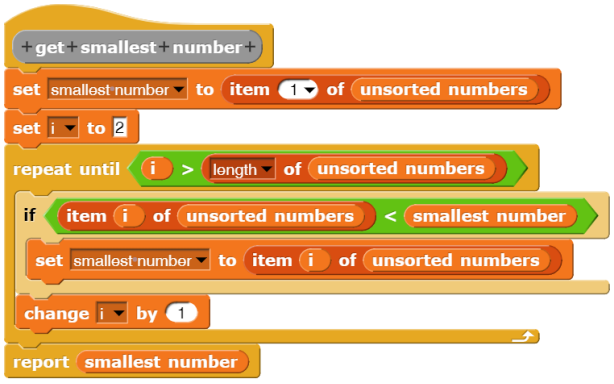
Wir erzeugen also eine Variable namens *unsorted numbers* und weisen der eine leere Liste zu. (Mithilfe der Pfeiltasten im *list*-Block könnten wir auch Anfangswerte in den dann entstehenden freien Plätzen vorgeben. Dabei ist der Typ der eingefügten Größen egal: Listen können alles aufnehmen, und das in bunter Reihenfolge.) Wird die Variable erzeugt, dann erscheint sie als *Watcher* auf der Bühne. Dort können wir im Kontextmenü unterschiedliche Darstellungsformen wählen oder die Liste als *dialog* beliebig im *Snap!*-Fenster positionieren.



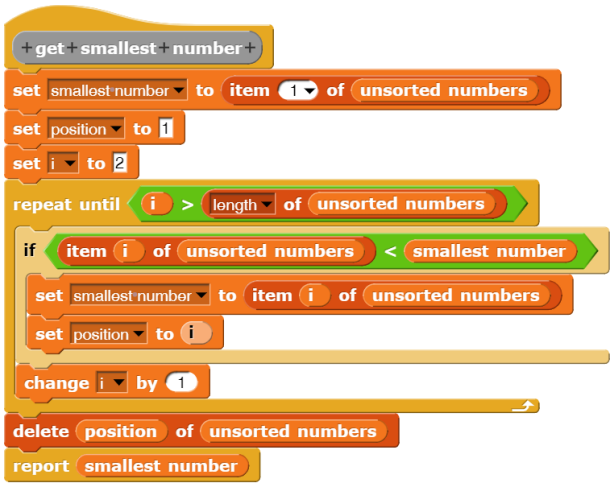
Auf die gleiche Art erzeugen wir eine zweite Liste sorted numbers, die später die sortierten Daten aufnehmen soll. Zuerst einmal brauchen wir unsortierte Daten – wie üblich Zufallszahlen. Die erzeugen wir mit einem kleinen Skript, wobei sich die Anzahl der Zahlen aus der Zahl der Wiederholungen in der Schleife ergibt.

Das Skript probieren wir mehrfach aus – immer wieder erhalten wir eine neue Zahlenliste. Toll! Voller Stolz bilden wir einen neuen Block namens *generate new numbers*. (Rechtsklick auf die Skript-Ebene.) In diesem hängen wir unser Skript einfach an den „Hut“ mit dem Blocknamen an. Fertig – wir haben einen neuen Befehl geschrieben! Den finden wir unten in der *Variables*-Palette – wenn wir nichts anderes angegeben haben.

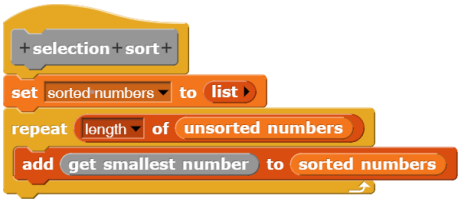
Aus dieser Zahlenliste wollen wir jetzt die kleinste Zahl heraussuchen. Dafür nehmen wir mal an, dass die erste Zahl schon die kleinste ist. Danach sehen wir alle folgenden Zahlen an. Ist eine kleiner als die bisherige kleinste Zahl, dann merken wir uns die. Sind wir durch, dann „reporten“ wir das Ergebnis – wir schreiben also eine Funktion *get smallest number*.



Das klappt auch prima. Allerdings nur einmal, denn die nächst-kleinere Zahl finden wir auf diese Weise nicht. Das geht erst, wenn wir jedes Mal die gefundene kleinste aus der Liste entfernen. Weil wir erst nach dem gesamten Durchlauf wissen, welches die kleinste Zahl war, merken wir uns neben deren Wert auch deren Platz – und werfen sie nach dem Durchlauf durch die Liste raus.



Das Sortieren einer Liste ist jetzt ganz einfach: Wir holen nacheinander die jeweils kleinste Zahl aus der unsortierten Liste und packen sie in die sortierte. Fertig. Das Skript verpacken wir wieder in einem neuen Block, den wir *selection sort* nennen.



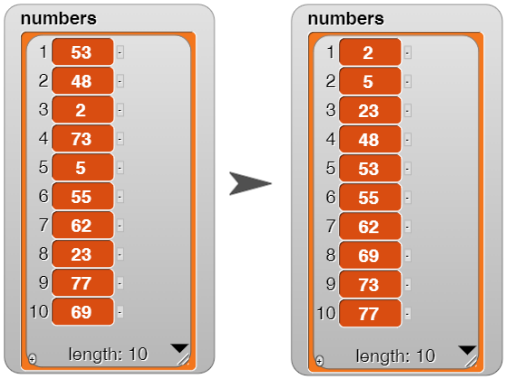
7.2 Sortieren mit Listen – Quicksort

Altersstufe: *Sekundarstufe II* Material: *Quicksort*

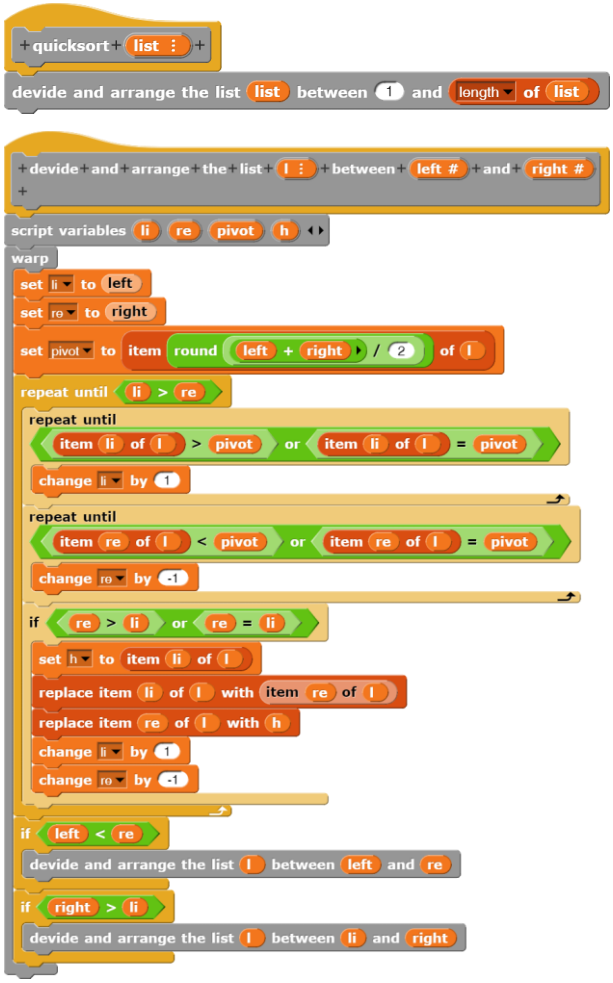
Als zweites, rekursives, Beispiel wollen wir Quicksort[42] in der gleichen Umgebung wie oben realisieren. Dafür schreiben wir uns erst mal eine etwas elegantere Methode zur Erzeugung neuer Zahlen, die einen Parameter und lokale Skriptvariable benutzt. Damit können wir angeben, wie viele Zahlen wir haben wollen. Um auch mit größeren Zahlenmengen umgehen zu können, verpacken wir alles in einen *Warp*-Block.

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

Die eigentliche Arbeit erfolgt im Block *devide and arrange the list <l> between <left> and <right>*. Als Pivot-Element wählen wir dort das mittlere der jeweiligen Teilliste.



10000 Zufallszahlen können wir damit in ca. 2 Sekunden sortieren.

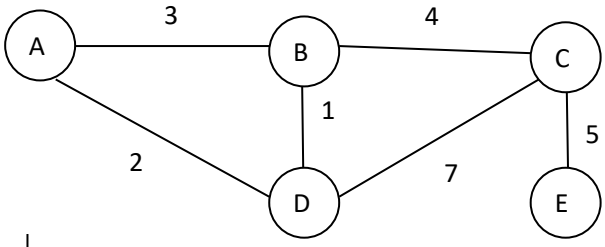


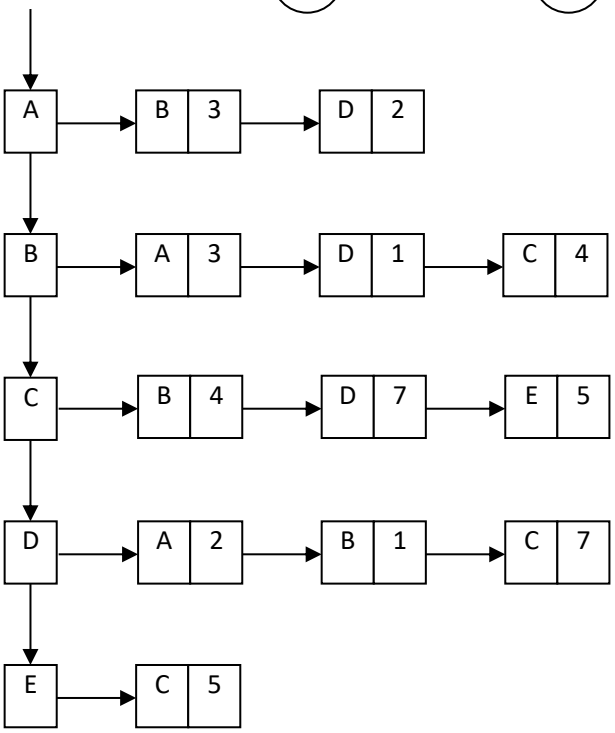
[42] Das Verfahren findet sich in diversen Versionen im Internet, z. B. unter http://de.wikipedia.org/wiki/Quicksort. Hier wurde eine In-place-implementierung gewählt.

7.3 Kürzeste Wege mit dem Dijkstra-Verfahren

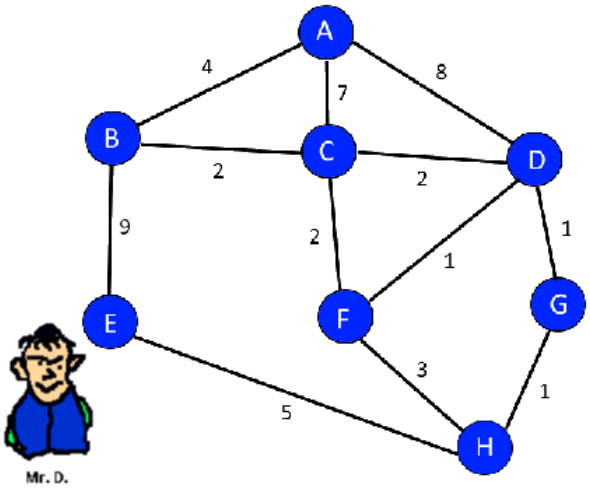
Altersstufe: *Sekundarstufe II* Material: *Dijkstra routing*

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.

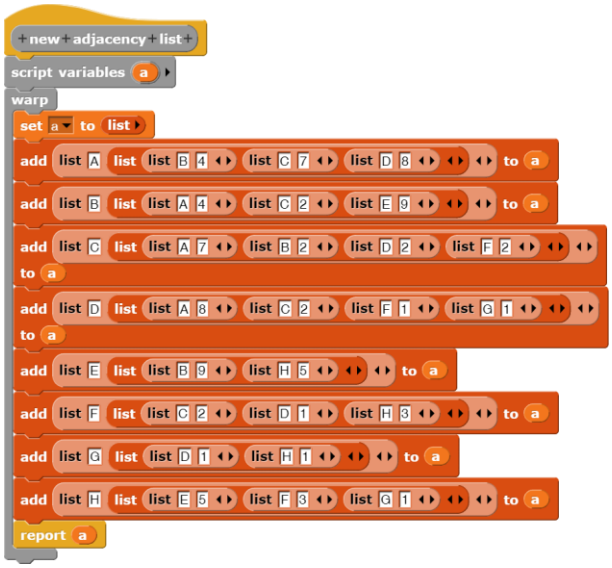




Zur Bearbeitung des Problems benötigen wir natürlich einen Spezialisten: wir zeichnen *Mr. D*. Dieser muss in der Lage sein, die Adjazenzliste eines gegebenen Graphen zu erzeugen. Den Graphen zeichnen wir einfach auf den Hintergrund – hier sehr geschmackvoll geschehen.

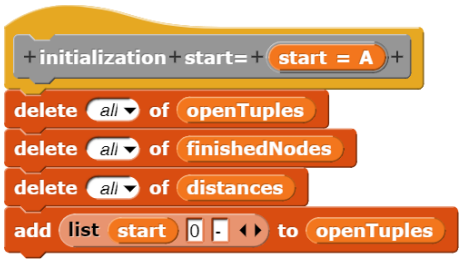


Die Liste erzeugen wir statisch durch Einfügen der entsprechenden Elemente in eine lokale Liste, die wir als Ergebnis der Operation zurückgeben.

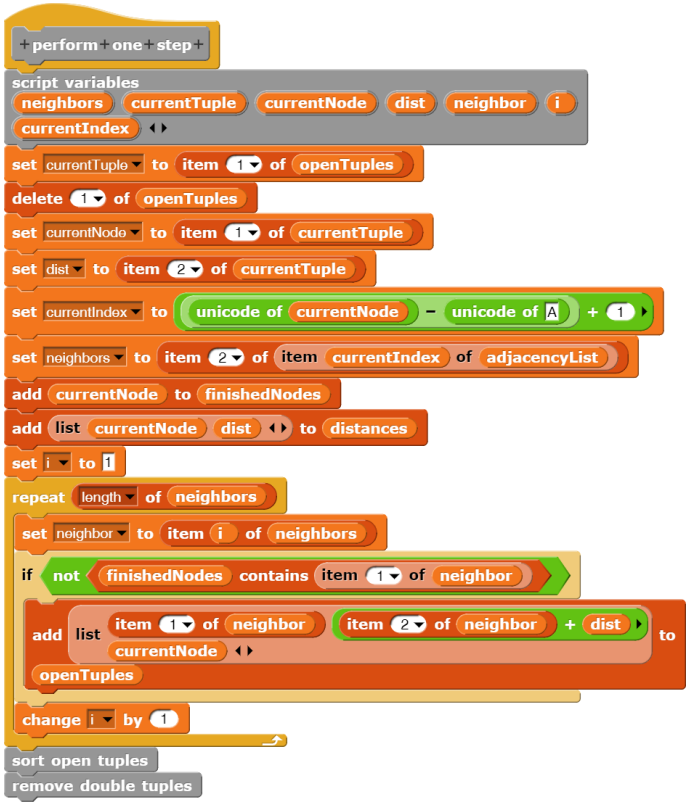


Die globale Variable *adjacencyList* erhält dann diese Werte über eine einfache Zuweisung.

Zur weiteren Bearbeitung benötigen wir noch drei andere Listen: Die Liste der *openTuples* nimmt Tupel auf, die den Namen des Knotens, seine Gesamtentfernung vom Startknoten und den Namen des Vorgängerknotens enthalten; die Liste *distances* nimmt Tupel auf, die den Namen des Knotens und seine Gesamtentfernung vom Startknoten enthalten, sie wird bei Neueintragungen jeweils neu sortiert, sodass der Knoten mit der kürzesten Entfernung vom *Start* vorne steht; die Liste *finishedNodes* enthält die Namen der Knoten, die bereits fertig bearbeitet wurden. Die Einrichtung dieser Listen für den Start fassen wir in einer Methode *initialization* zusammen, der auch der Name des Startknotens übergeben wird. Nach ihrem Aufruf ergibt sich das folgende Bild.



Die Wegsuche ist in dieser Version relativ einfach, da der größte Teil der „Intelligenz“ in den Umgang mit den Listen gesteckt wurde. Dieser erfolgt in der Methode *perform one step*.



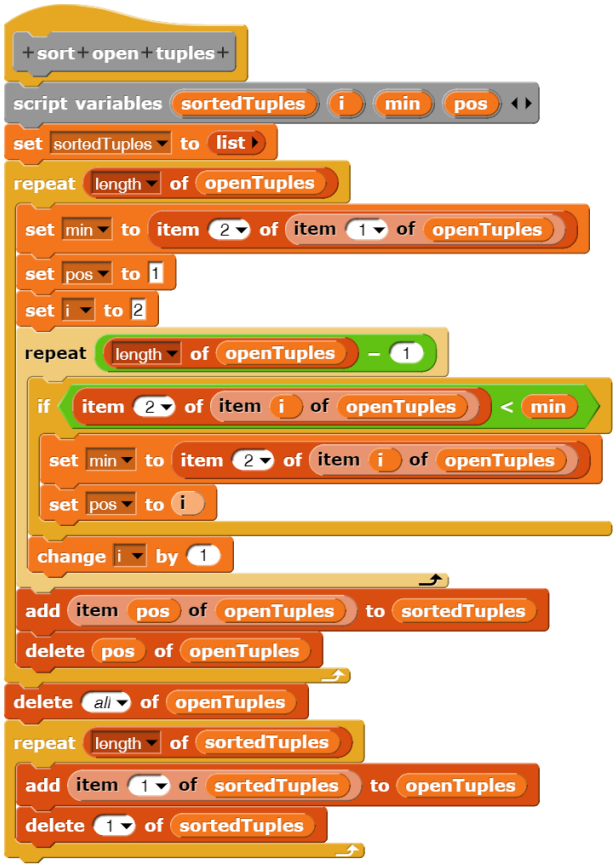
Für das Tupel currentTupel mit der kleinsten Entfernung werden für die Nachbarknoten die neuen Entfernungen berechnet.

Dann wird der Knoten als bearbeitet markiert und alle unbearbeiteten Nachbarn mit neuer Gesamtentfernung und Vorgängerknoten in openTupels eingetragen.

Diese Liste wird nach der Entfernung sortiert und Tupel mit größeren Entfernungen werden gelöscht.

Bis auf die drei Hilfsmethoden ist das Routing jetzt fertig:

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



die Liste sortedTuples *nimmt die sortierten Tupel auf*

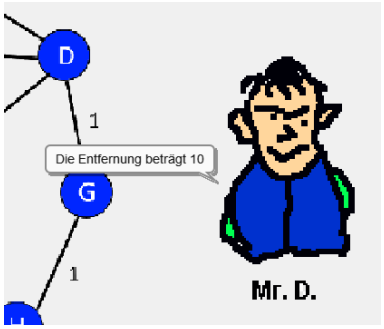
Annahme, dass die kleinste Entfernung ganz vorne steht.

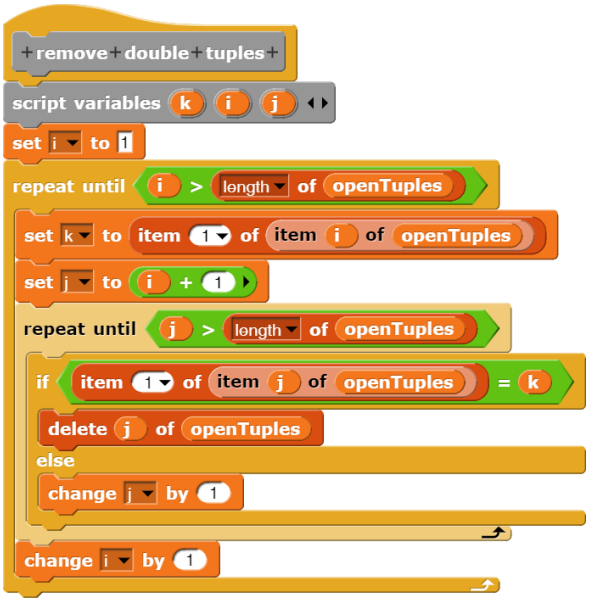
ggf. noch kleinere Entfernungen finden

das Tupel mit der kleinsten Entfernung zu sortedTuples *hinzufügen und in* openTuples *löschen*

zuletzt die sortierte Liste zurück kopieren

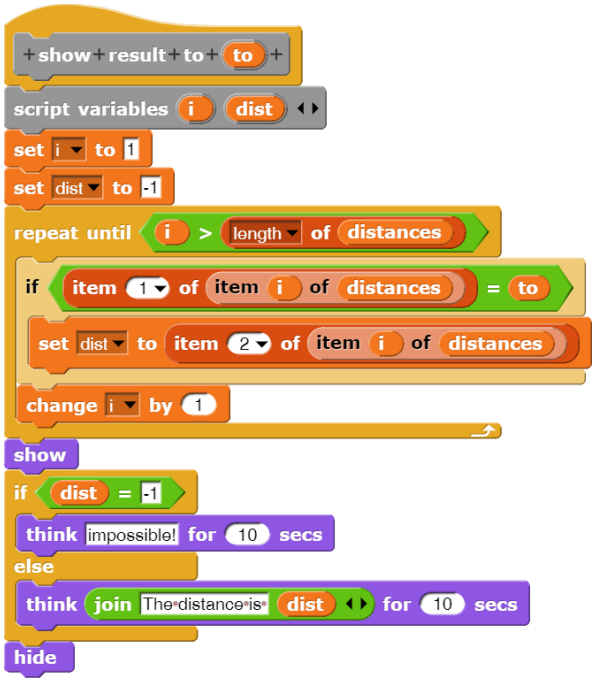
Jetzt steht für jeden Knoten das Tupel mit der kleinsten Entfernung vorne in der Liste. Falls noch andere Tupel für diesen Knoten auftreten, werden sie gelöscht. Dann müssen wir nur noch die Entfernung zum gesuchten Knoten aus der Liste *distances* heraussuchen und von Mr. D. anzeigen lassen.





7.4 Matrizen und eigene Zählschleifen

Altersstufe: *Sekundarstufe II* Material: *Matrices*



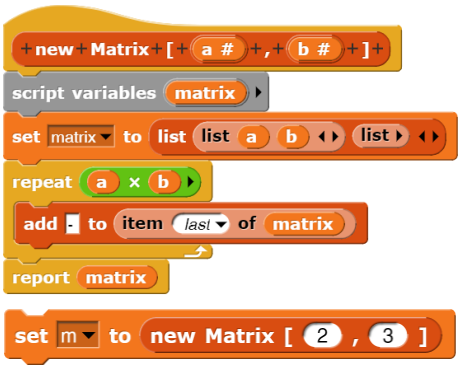
Wenn wir Listen mit direktem Zugriff auf jedes Element haben, dann benötigen wir eigentlich keine speziellen Reihungen, Stapel, Schlangen usw. Alle höheren Datenstrukturen lassen sich aus Listen aufbauen. Trotzdem basteln wir uns die Datenstruktur *matrix*, weil sie traditionell z. B. bei den Adjazenzmatrizen Verwendung findet. *(Achtung: der Kürze halber verzichten wir auf alle Sicherheitsabfragen!)*

Wir verpacken eine Matrix natürlich in einer Liste. Dafür vereinbaren wir (willkürlich) die folgende Listenstruktur:

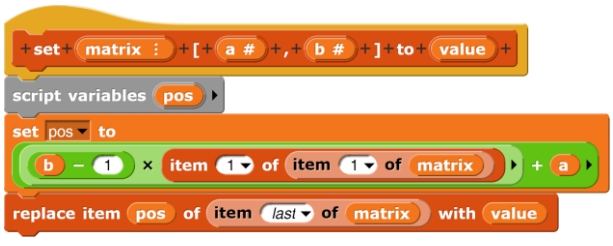
[[Liste mit Größen der Indexbereiche] [Liste mit Daten……]]

Die Dimension der Matrix ergibt sich dann direkt aus den Einträgen der ersten Teilliste. Eine zweidimensionale Reihung mit jeweils zwei Werten pro Zeile hätte die folgende Struktur: *[[2,3] [1,2,3,4,5,6]]*

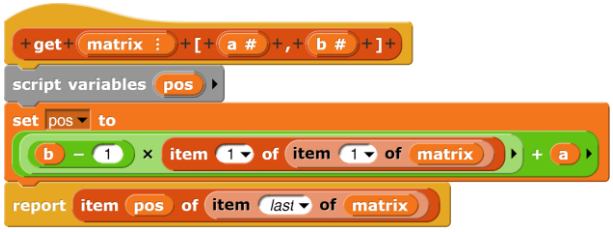
Wir legen eine zweidimensionale Matrix der Größe *a x b* an, indem wir die beiden gewünschten Listen erzeugen. Die erste enthält die beiden übergebenen Parameter, die zweite soll als leer gekennzeichnet sein, z. B. durch ein Minuszeichen für jedes Element. Das Ergebnis geben wir zurück. Wir benutzen globale Methoden, die wir der Listen-Palette zuordnen. Die Syntax kann völlig frei gewählt werden, zum Beispiel auch mit Klammern, wenn man das mag.



Jetzt schreiben wir mit *set* Werte in die Matrix, schön übersichtlich. Wir berechnen den Platz der zu ändernden Stelle mithilfe der Dimensionen. Dann überschreiben wir den entsprechenden Eintrag.



Zum Lesen von Matrixeinträgen dient die Methode *get*.



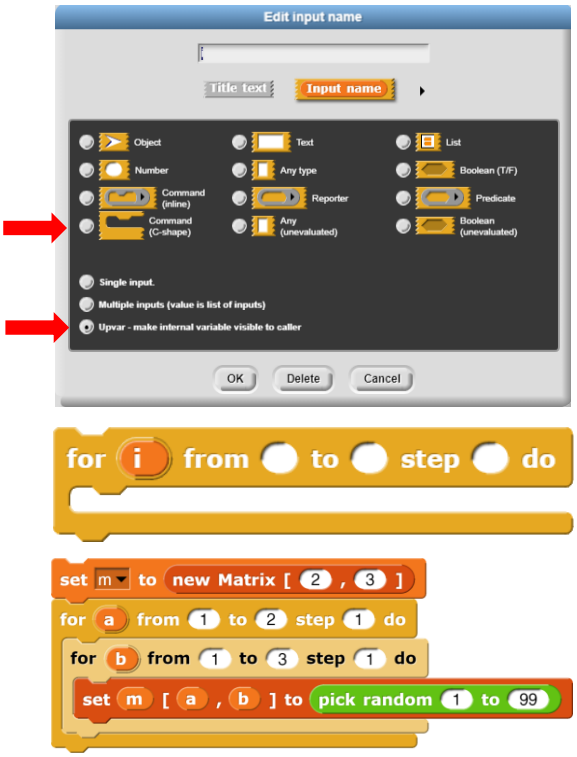
In vielen Programmiersprachen ist die Zählschleife das gängige Werkzeug, um Matrizen zu durchlaufen. In *Snap!* finden wir so etwas in der *Control*-Palette, aber wir können eine solche Kontrollstruktur auch selbst schreiben, z. B. um sie mit einer Schrittweite auszustatten. Dazu erzeugen wir einen neuen Block *for <laufvariable> from <start> to <end> step <step> do <skript>* und sehen uns die Art der Parameter etwas genauer an.

Die Laufvariable *i* kennzeichnen wir als *upvar*. Damit kann ihre Bezeichnung „nach außen“ geändert werden, obwohl ihr interner Name gleichbleibt – eben *i*.

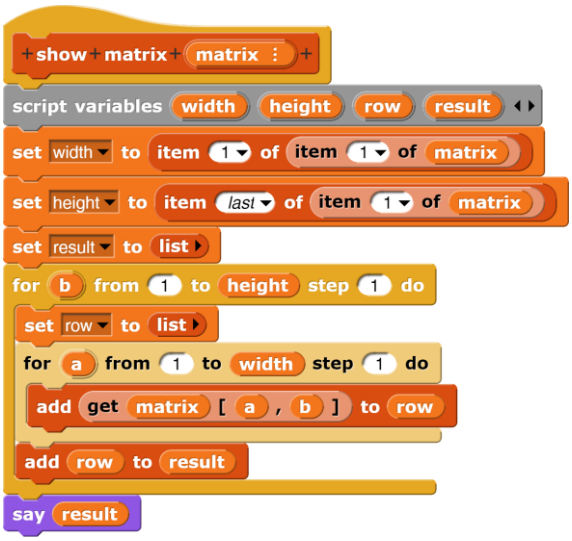
start, end und *step* sind normale Zahlenparameter.

Das Skript kennzeichnen wir als *C-shaped command*. Damit wird es als Befehlsfolge angesehen, die dem Block unverändert übergeben, also nicht evaluiert wird.

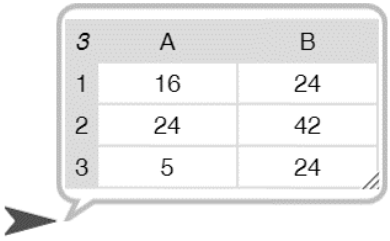
C-shaped sorgt dafür, dass der Block das übliche Aussehen von *Snap!*-Kommandos erhält, bei denen die auszuführende Befehlsfolge in das „Maul“ des Cs eingefügt wird.



Mithilfe dieser Schleifenart können wir eine Matrix schnell mit Zufallszahlen füllen.

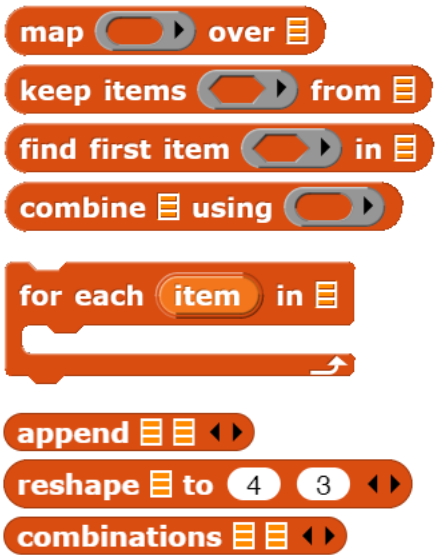


Zuletzt wollen wir die Matrix „anständig“ am Bildschirm darstellen, also in der üblichen zweidimensionalen Tabellenform. Dazu erzeugen wir eine Liste, die mit Teillisten, den Zeilen der Matrix, gefüllt wird, die die Tabellendaten enthalten. Diese Liste wird anschließend angezeigt und kann dann auch als *Table view* überall hin verschoben werden.



3	A	B
1	16	24
2	24	42
3	5	24

7.5 Höhere Listenoperationen



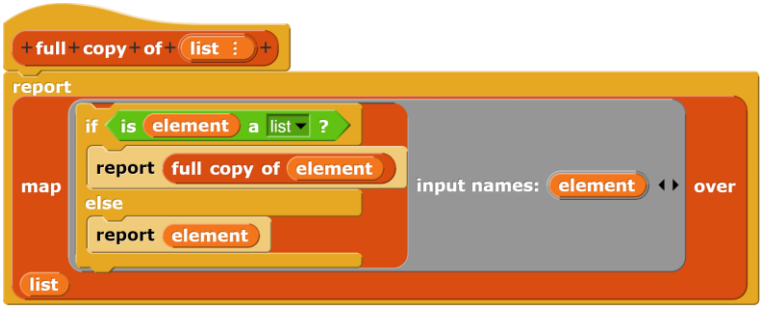
In der Variable-Palette finden wir einige sehr schnelle Blöcke, die komplexere Operationen auch auf großen Listen zulassen. Der wichtigste davon ist der *map … over…*-Block. Er wendet ein Skript, das sich in dem grauen Ring befindet, der Reihe nach auf alle Elemente einer Liste an und gibt die Ergebnisse als neue Liste zurück. Im Default-Fall wird das aktuelle Listenelement in einen leer gelassenen Platzhalter im Skript eingesetzt. Möchte man es lesbarer gestalten, dann vergibt man einen Namen für das Element und verwendet den im Skript, benötigt man den Index des Listenelements, dann erhält man den im Feld nach dem Elementnamen. Danach findet man noch eine Referenz zur Gesamtliste.

Als Beispiel wollen wir die Kopie einer Liste erzeugen. Die erste Liste soll einfach aus den ersten 100000 natürlichen Zahlen bestehen.

Von dieser Liste können wir jetzt auf unterschiedliche Weise Kopien erzeugen. Im einfachsten Fall, indem wir die *map…over…*-Funktion direkt anwenden. Wir können aber auch das altuelle Listenelement benennen und es unter dem Namen zurückgeben, und wir können das auch explizit mithilfe des *report*-Blocks machen.

Natürlich können wir auch eine dieser Versionen innerhalb eines neuen Blocks zum Kopieren von einfachen Listen verwenden.

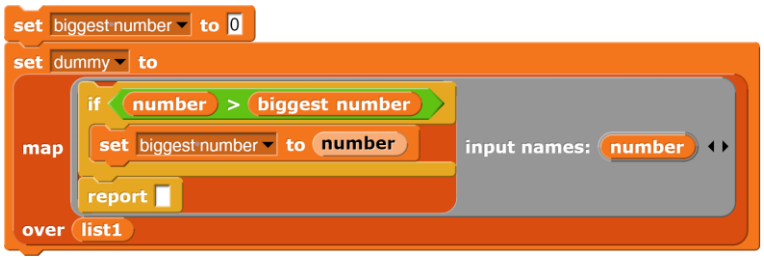
Und wenn man sich daran erinnert, dass Listen auch weitere Teillisten enthalten können, dann müssen natürlich auch diese bei einer Kopie gesondert kopiert werden – schön rekursiv.



In manchen Fällen möchte man die Schnelligkeit des *map…over…*-Blocks ausnutzen, um eine Liste zu durchsuchen. Als Beispiel wollen wir das größte Element einer Liste finden. Da beim *map…over…*-Block für jedes Element ein neues Listenelement zurückgegeben werden muss, können wir nicht einfach das gesuchte größte Element als Resultat ermitteln. Stattdessen lassen wir die Liste durchlaufen und ermitteln das größte Element als *Seiteneffekt* gesondert. Die Ergebnisse des eigentlichen Funktionsaufrufs ignorieren wir, z. B. indem wir sie einer Variablen *dummy* zuweisen und nicht weiter auswerten.

Als erstes benötigen wir unsortierte Zahlen, die wir schnell mithilfe des *map…over…*-Blocks erzeugen.

Danach suchen wir die größte Zahl aus diesen 100000 Werten heraus, indem wir die Liste durchlaufen und dabei die jeweils größte Zahl ermitteln. Als Elemente der Resultatliste geben wir einfach „nichts“ zurück.

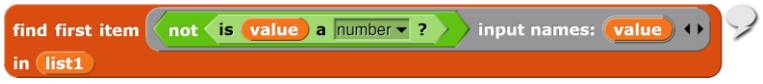


Eine spezielle Kontrollstruktur zum Durchlaufen einer Liste ist der Block *for each…in…*. In den Bibliotheken findet man auch eine Version, die Zugriff auf den Index gestattet. Auch mit diesem Block können wir die größte Zahl einer Liste ermitteln. Schnell geht das bei langen Listen aber nur, wenn wir den *warp*-Block einsetzen – dann aber sehr schnell.

Ein weiterer sehr nützlicher Block ist der *keep items…from…*-Block. Er enthält im grauen Ring ein *Prädikat*, also eine Funktion, die entweder *true* oder *false* ergibt, das auf alle Listenelemente angewendet wird. Das Ergebnis besteht aus einer Liste, die nur die als *true* ausgewerteten Listenelemente enthält.

Wir nehmen wieder die eben erzeugte Liste mit 100000 Zufallszahlen und wollen aus dieser nur diejenigen heraussuchen, die gerade, also durch 2 teilbar sind. Auch hier bekommen wir über die kleinen schwarzen Pfeile rechts am grauen Ring Zugriff auf das Listenelement, seinen Index und die Gesamtliste.

Der Block *find first item …in…* arbeitet sehr ähnlich. Er sucht das erste Element heraus, das dem angegebenen Prädikat entspricht. Findet er kein Element, dann gibt er folgerichtig „nichts“ zurück. Der Block ist z. B. nützlich, um sicherzustellen, dass sich keine „unzulässigen“ Elemente in einer Liste befinden. Wollen wir z. B. arithmetische Operationen mit den Listenelemente ausführen, dann sollten sich auch nur Zahlen in der Liste befinden.

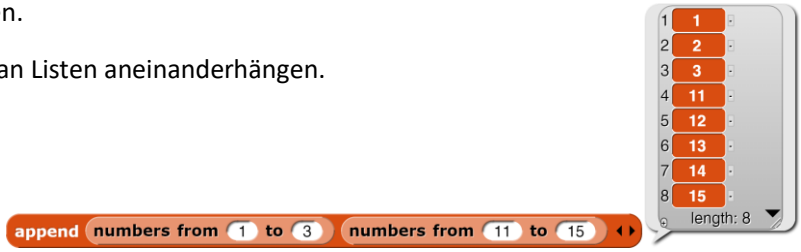


Möchte man eine arithmetische, logische oder listenbezogene Operation mit allen Listenelemente durchführen, dann benutzt man den *combine …using …*-Block. Er wendet sukzessive den angegebenen zweistelligen Operator auf die Gesamtliste an. Als Beispiel wollen wir die Summe der Listenelemente von *liste1* berechnen.

Enthält die Liste Zeichenketten, dann können wir daraus ein langes Wort bilden.

Und natürlich können wir die höheren Listenoperationen auch kombinieren, z. B. indem wir die Summe aller Vielfachen von 231 einer Liste berechnen.

Mithilfe des *append*-Blocks kann man Listen aneinanderhängen.



Und der *reshape*-Block „formatiert“ eine Liste auf andere Dimensionen um.

3	A	B	C	D	E
1	1	2	3	4	5
2	6	7	8	9	10
3	11	12	13	14	15

Der *combinations*-Block bildet das kartesische Produkt mehrerer Mengen. Er kombiniert (bei zwei Mengen) also alle Elemente der einen mit allen der anderen.

15	A	B
1	1	11
2	1	12
3	1	13
4	1	14
5	1	15
6	2	11
7	2	12
8	2	13
9	2	14
10	2	15
11	3	11
12	3	12
13	3	13
14	3	14
15	3	15

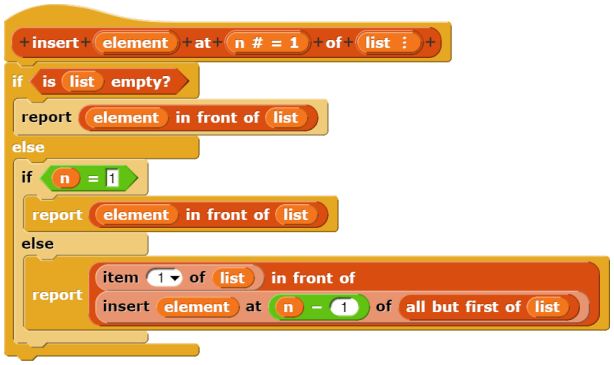
Und als letzte Anmerkung: die Möglichkeiten des …*of*…-Blocks sollte man bei Bedarf in Erwägung ziehen, z. B. um die Reihenfolge einer Liste umzukehren.

7.6 Rekursive Listenoperationen

Nach den sehr leistungsfähigen Blöcken wollen wir uns noch kurz mit den eher elementaren Blöcken für rekursives Programmieren beschäftigen. Aus diesen lassen sich alle erweiterten Operationen aufbauen. Nicht übermäßig effizient, aber elegant. Der erste Block gestattet es, ein Element vorne in eine Liste einzufügen, der zweite liefert das erste Element[43]. Der dritte Block liefert die Restliste nach dem ersten Element, und der letzte prüft, ob eine Liste leer ist.

Wir können Listen als Paare auffassen, die aus einem ersten Element und dem Rest bestehen, wobei diese Paarelemente auch leer sein können. Traditionell bezeichnet man sie als *car* (sprich „carr“) und *cdr* („cudder“). Als Beispiel für die Anwendung wollen wir die Länge einer Liste ermitteln.

Auf sehr ähnliche Weise können wir ein neues Listenelement an einem angegebenen Platz einer Liste einfügen. Ist die Liste leer, dann liefern wir einfach eine Liste, die nur das neue Element enthält. Ansonsten sehen wir nach, ob das Element vorne angefügt werden soll, und tun das ggf. Ist auch das nicht der Fall, dann liefern wir eine Liste, die vorne das schon bisher erste Element enthält und als Rest eine Liste, in die das neue Element einen Platz vor dem bisherigen, aber in der Restliste erhält.

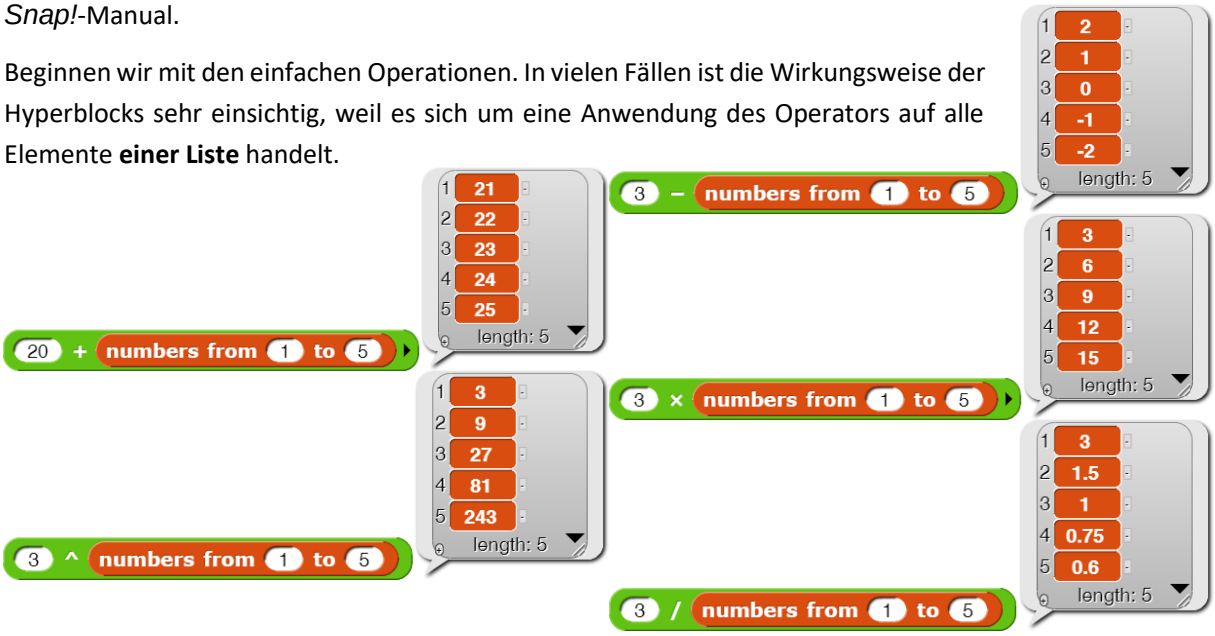


[43] Oder jedes andere, aber das vergessen wir mal. 😉

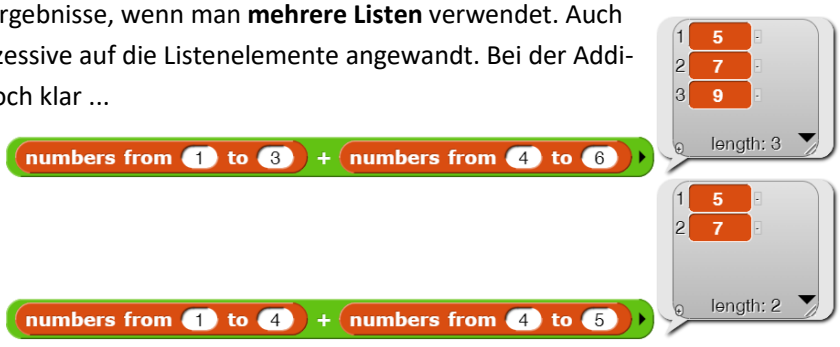
7.7 Hyperblocks

Einige der z. B. arithmetischen Operatoren wurden in *Snap!* so erweitert, dass sie auch auf Listen anwendbar sind. Das Ergebnis sind außerordentlich schnelle Listenoperationen, die z. B. die Manipulation von bewegten Bildern in Echtzeit erlauben. Hyperblocks können also auf große Datenmengen angewandt werden und sind deshalb eher für den Umgang mit Medien gedacht. Manche der Operationen sind direkt einleuchtend, manche aber ziemlich gewöhnungsbedürftig. Man sollte die Verfahren jeweils anhand kleiner Testlisten erproben, bevor man sie auf große Datenmengen „loslässt“. Obwohl viele der Operationen an mathematische Verfahren angelehnt sind, liefern sie oft keine mathematisch korrekten Ergebnisse, z. B. weil sie wegen unterschiedlicher Dimensionen gar nicht mathematisch zulässig sind. Will man aber z. B. eine Matrixmultiplikation implementieren (s.u.), dann bietet es sich an, die Dimensionen vorab zu überprüfen und danach die Hyperblocks arbeiten zu lassen. Eine ausführliche Beschreibung der Hyperblocks findet sich im *Snap!*-Manual.

Beginnen wir mit den einfachen Operationen. In vielen Fällen ist die Wirkungsweise der Hyperblocks sehr einsichtig, weil es sich um eine Anwendung des Operators auf alle Elemente **einer Liste** handelt.

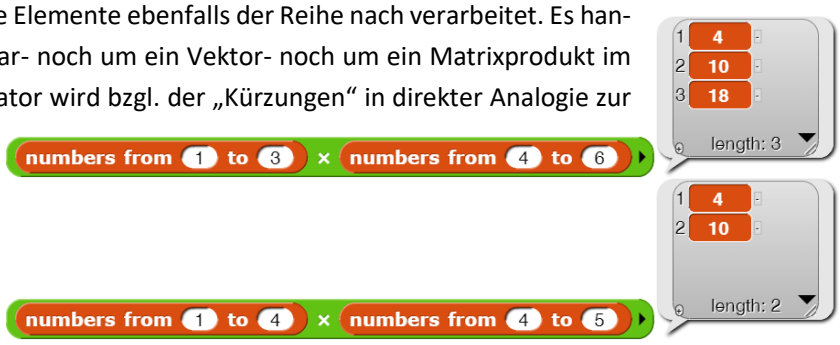


Etwas überraschender sind die Ergebnisse, wenn man **mehrere Listen** verwendet. Auch hier werden die Operatoren sukzessive auf die Listenelemente angewandt. Bei der Addition gleichlanger Listen ist das noch klar ...



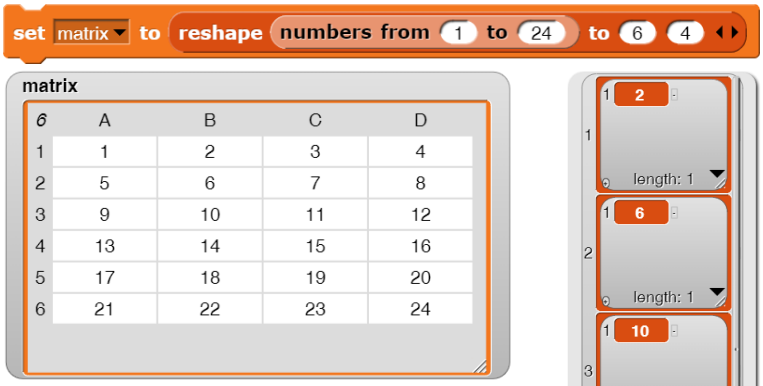
... aber bei unterschiedlich langen Listen muss man wissen, dass das Ergebnis gekürzt wird.

Bei der Multiplikation werden die Elemente ebenfalls der Reihe nach verarbeitet. Es handelt sich also weder um ein Skalar- noch um ein Vektor- noch um ein Matrixprodukt im mathematischen Sinn. Der Operator wird bzgl. der „Kürzungen“ in direkter Analogie zur Addition angewandt.



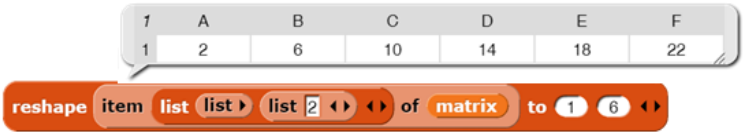
Verarbeitet man komplexere Listenstrukturen, dann sollte man vorher das Handbuch lesen, um deren Verarbeitung zu verstehen. Als Beispiel für die Verwendung leerer Listen als „Richtungs-Flag“ soll das Auslesen von Spalten einer Matrix dienen.

Zuerst werden die Listenelemente erzeugt und in Matrixform gebracht. Dabei ist zu beachten, dass der erste Parameter die Zeilenzahl, der zweite die Spaltenzahl bestimmt. Man erhält also eine 4 X 6-Matrix.

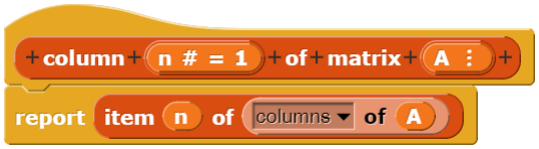


Aus dieser kann man die n-te Spalte unter Angabe der Spaltennummer extrahieren …

… und bei Bedarf umformatieren.

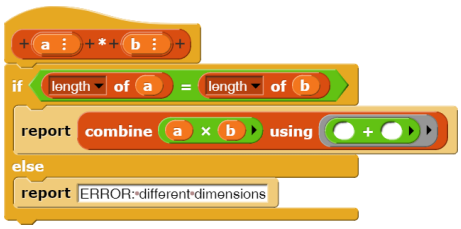


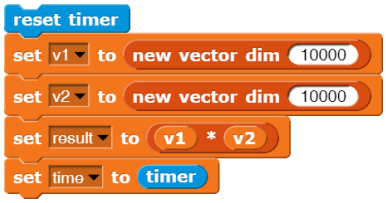
Sehr viel verständlicher für die Spaltenbestimmung ist die Berechnung der transponierten Matrix, von der dann die n-te Zeile genommen wird:



Mit diesen Kenntnissen kann man schon etwas anfangen. Wir bauen zuerst einmal einen Block für das *Skalarprodukt* zweier Vektoren. Natürlich benötigen wir zum Testen unterschiedlich lange Vektoren, hier mit Zufallszahlen als Inhalten. Wir nutzen dafür den schnellen *map…over…*-Block.

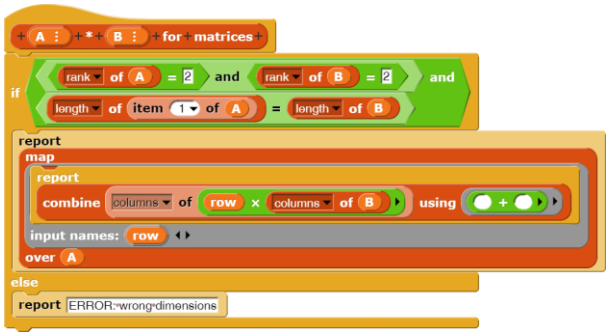
Zur Berechnung des skalaren Produkts nutzen wir den Multiplikations-Block als Hyperblock und addieren die Ergebnisse mithilfe des *combine…using…*-Reporters. Und weil wir mathematisch korrekt bleiben wollen, überprüfen wir vorher die Dimensionen der Vektoren. Das Ergebnis ist ein sehr schneller Block, der z. B. für zwei 10000-stellige Vektoren die gemessene Multiplikationszeit von 0 Sekunden ergibt.



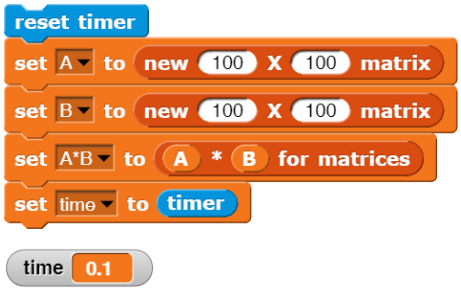


Wenn das für Vektoren so gut klappt, dann versuchen wir uns natürlich auch an einer *Matrix-Multiplikation* in der üblichen Form. Zuerst einmal müssen wir Matrizen erzeugen. Das machen wir ähnlich wie bei den Vektoren.

Der Block für die Matrix-Multiplikation überprüft zuerst natürlich ebenfalls, ob die Dimensionen stimmen. Danach nutzt er Hyperblocks und höhere Funktionen. Weil *columns of <liste>* die transponierte Matrix berechnet, können wir die Zeilen der ersten Matrix *A* jeweils mit allen Zeilen der transponierten Matrix *B*, also mit den Spalten von *B*, skalar multiplizieren.



Auch hier können wir die Zeit für größere Datenmengen messen. Multiplizieren wir zwei 100X100-Matrizen, dann dauert das 0.1 Sekunden.



A

4	A	B
1	10	10
2	7	10
3	9	3
4	8	9

B

2	A	B	C	D	E
1	5	9	3	6	9
2	1	6	2	8	10

A*B

4	A	B	C	D	E
1	60	150	50	140	190
2	45	123	41	122	163
3	48	99	33	78	111
4	49	126	42	120	162

7.8 Schnelle Bildmanipulation mit vorkompilierten Blöcken

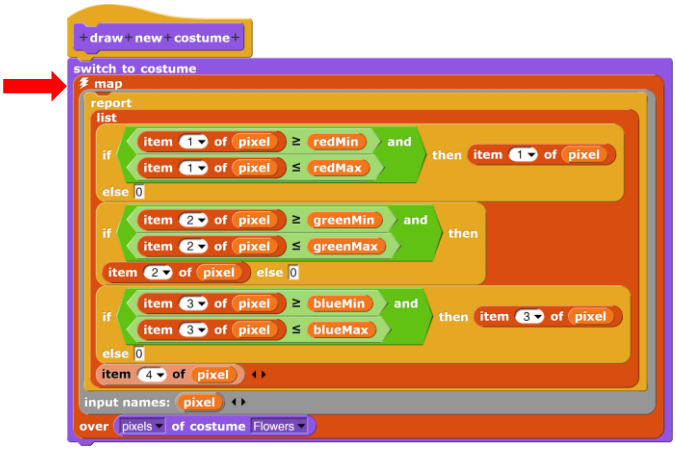
Altersstufe: *Sekundarstufe II* Material: *Expose flowers*

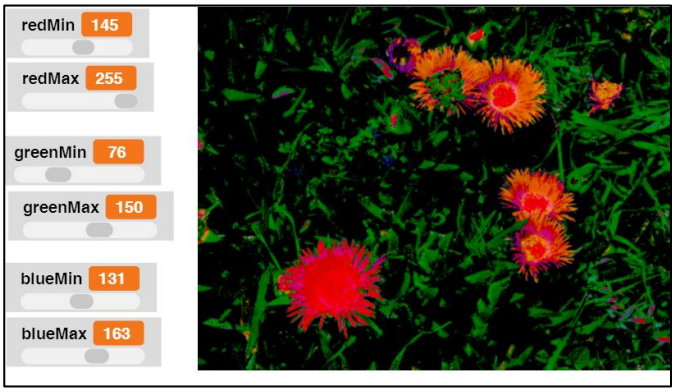
Als letzte Anwendung wollen wir zeigen, wie man mit dem vorab kompilierten *map…over…*-Block ein Bild in Echtzeit ändern und anzeigen kann. Als Beispiel wählen wir ein Farbbild, in dem wir nur einstellbare Farbbereiche anzeigen wollen. Damit können z. B. Gesichter in einem Bild identifiziert oder, wie hier, Blumen extrahiert werden.

Um direkt auf Änderungen reagieren zu können, benutzen wir für die Grenzen der Farbbereiche jeweils zwei Variable: den aktuellen Wert und den letzten Wert. Ändert sich der aktuelle Wert, dann wird das Bild neu berechnet und der letzte Wert des Farbwerts wird angepasst. Für die Änderung der Variablenwerte benutzen wir die Slider-Darstellung der Variablen.

Zu Beginn erhalten die „alten Werte“ einfach die aktuellen Werte. Danach reagierten die Skripte wie beschrieben auf Änderungen, z. B. für den roten Bereich.

Die Neuberechnung des Bildes erfolgt, indem für die drei Farbkanäle jeweils überprüft wird, ob der Farbwert innerhalb des gewählten Bereichs liegt. Ist das der Fall, dann wird der Farbwert übernommen, sonst auf Null gesetzt. Zuletzt wird der aktuelle Transparenzwert an das Pixel angehängt. Zu beachten ist der kleine Blitz oben im *map*-Block. Er bedeutet, dass das Skript innerhalb des Blocks vorkompiliert wird und deshalb sehr schnell auf alle Listenelemente angewendet werden kann. Man erhält diese Option durch Auswahl im Kontextmenüs des Blocks.





7.9 Aufgaben

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

2. Ergänzen Sie die angegebenen Methoden so, dass **Fehleingaben** abgefangen werden.

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

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

5. a: Implementieren Sie die Datenstruktur **Stapel**.
   b: Implementieren Sie die Datenstruktur **Schlange**.

6. 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

7. Implementieren Sie andere **Kontrollstrukturen**:
   a: do <script> until <prädikat>
   b: while <prädikat> do <script>
   c: case <variable> of < [[wert1,script1], [wert2,script2], [wert3,script3], …] >

8. Implementieren Sie **rekursiv** nur unter Verwendung der elementaren rekursiven Operationen
   a: die Datenstrukturen Stapel und Schlange.
   b: eine Operation, die das n-te Element einer Liste löscht.
   c: eine Operation, die das n-te Element einer Liste durch ein neues Element ersetzt.
   b: eine Operation, die das n-te Element einer Liste als Ergebnis liefert.
   b: eine Operation, die ein neues Element am Ende einer Liste anfügt.

9. Implementieren Sie die üblichen Matrixoperationen mithilfe von Hyperblocks.

10. Implementieren Sie ein Verfahren, um den Kontrast eines Bildes zu erhöhen, mithilfe von Hyperblocks.

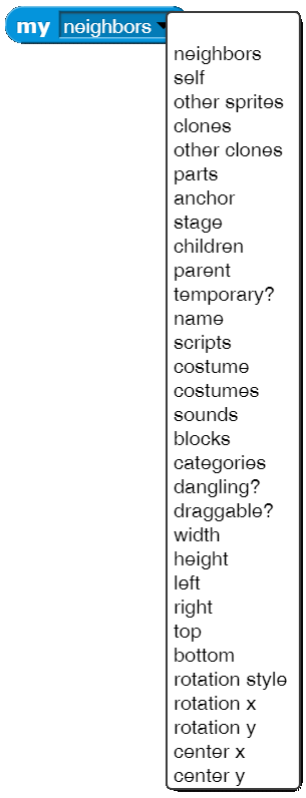
11. a: Implementieren Sie ein Verfahren, um ein Farbbild in ein Schwarzweißbild umzuwandeln, mithilfe eines vorkompilierten Blocks.
    b: Implementieren Sie ein Verfahren, um drei Farbauszüge in den Grundfarben zu erhalten, mithilfe von vorkompilierten Blöcken.

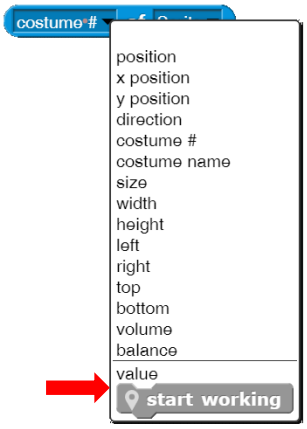
8 Objektorientierte Programmierung

Auch bisher wurden OOP-Methoden benutzt – weil es kaum anders geht. An dieser Stelle wollen wird die OOP-Möglichkeiten von *Snap!* etwas ausführlicher darstellen. Dabei verweisen wir ausdrücklich auf das *Reference-Manual* von *Snap!*, in dem die Verfahren kompakt erläutert werden. Man findet es, wenn man oben-links auf das *Snap!*-Symbol klickt.

Die für die OOP bedeutsamen Blöcke finden wir in der *Control*- und der *Sensing*-Palette, aber auch das Kontextmenü im Sprite-Bereich ist zu beachten. Die unteren Blöcke der *Control*-Palette dienen zur „dynamischen" Verwaltung von Sprites, das Menü zur „statischen". Dieser Unterschied ist bedeutsam, weil davon ausgegangen wird, dass nur die statischen Klone von Dauer sein sollen, die anderen werden z. B. beim Speichern gelöscht und im Sprite-Bereich gar nicht erst angezeigt.

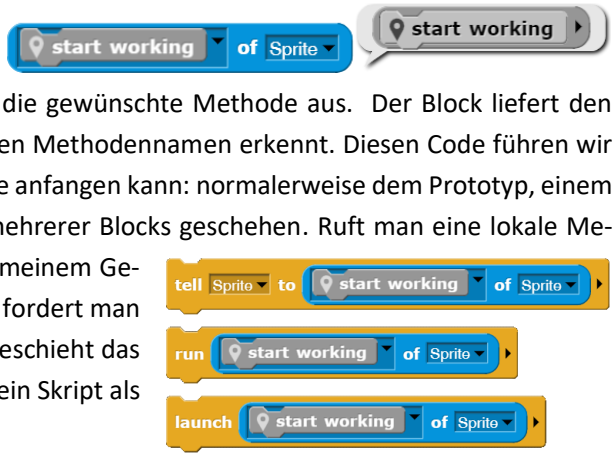
Snap! arbeitet natürlich die ganze Zeit mit Objekten, die hier *Sprites* genannt werden. Sie verfügen über eigene Attribute (*Position, Richtung, Kostüm, …*), auf die mithilfe unterschiedlichen Blöcke zugegriffen werden kann. Der *my <attribute>* - Block liefert die ganze Palette, der *<attribute> of <sprite>* - Block kennt die wichtigsten und zeigt darüber hinaus die lokalen Variablen und Methoden eines Sprites an.



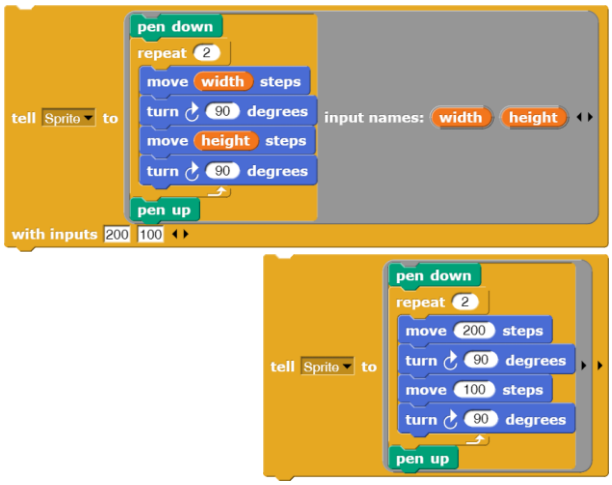


Man erhält den Wert einer lokalen Variablen (hier: der Position) eines anderen Sprites also z. B. mit

Zur Auswahl einer lokalen Methode setzen wir den Prototyp des betrachteten Objekts rechts in den *<attribut> of <sprite>*-Block ein und wählen dann die gewünschte Methode aus. Der Block liefert den *Code* der Methode, was man am grauen Ring um den Methodennamen erkennt. Diesen Code führen wir im Kontext eines Sprites aus, das etwas mit dem Code anfangen kann: normalerweise dem Prototyp, einem Klon oder einer Kopie von ihm. Das kann mithilfe mehrerer Blocks geschehen. Ruft man eine lokale Methode eines Sprites „von außen“ auf, dann ist nach meinem Geschmack der *run*-Block am intuitivsten zu verstehen, fordert man ein Sprite auf, globale Methoden aufzurufen, dann geschieht das besser durch den *tell*-Block. Der *lauch*-Block startet ein Skript als unabhängigen Prozess.

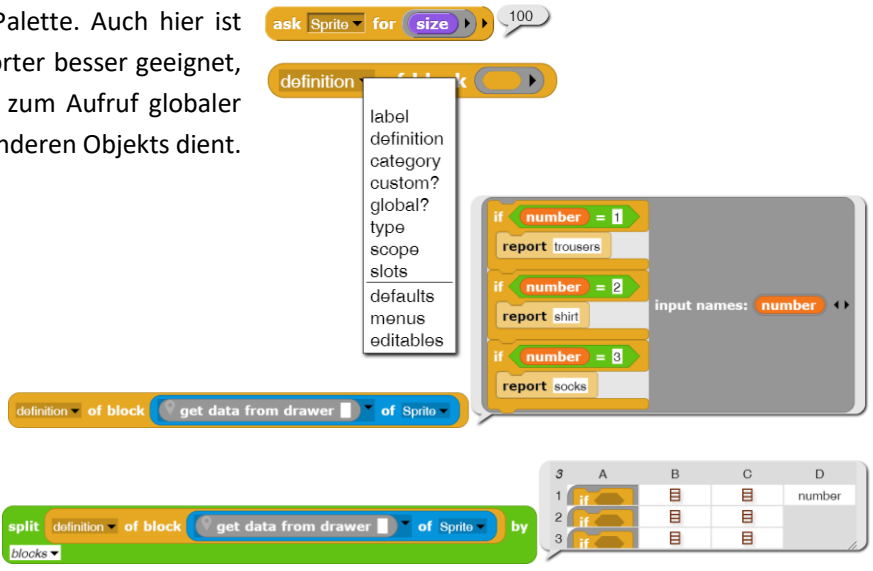


Da in einen grauen Ring ein Skript eingefügt wird, kann dieses natürlich auch aus mehreren Befehlen bestehen, und es können auch Parameter benutzt werden, die in die leeren Slots der Blöcke eingefügt werden und die man bei Bedarf benennen kann. Das kann z. B. sinnvoll sein, wenn man mehrere Parameter verwendet und sicherstellen will, dass diese an den richtigen Stellen eingesetzt werden. Da die Parameter außerhalb des aufgerufenen Sprites bestimmt werden, müssen sie auch (meist) außerhalb des Skripts unter *with inputs* aufgezählt werden. Werden die Parameter nicht benannt, dann werden sie der Reihe nach in leere Slots der Skriptblöcke eingesetzt. In vielen Fällen kann man bei globalen Blöcken Parameter direkt einsetzen.

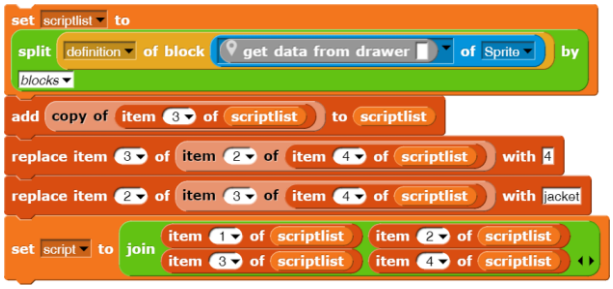


Lokale *Reporter*-Blöcke werden ganz ähnlich behandelt, allerdings von den entsprechenden *Reporter*-Blöcken der *Control*-Palette. Auch hier ist der *call*-Block für lokale Reporter besser geeignet, während der *ask*-Block eher zum Aufruf globaler Methoden im Kontext eines anderen Objekts dient.

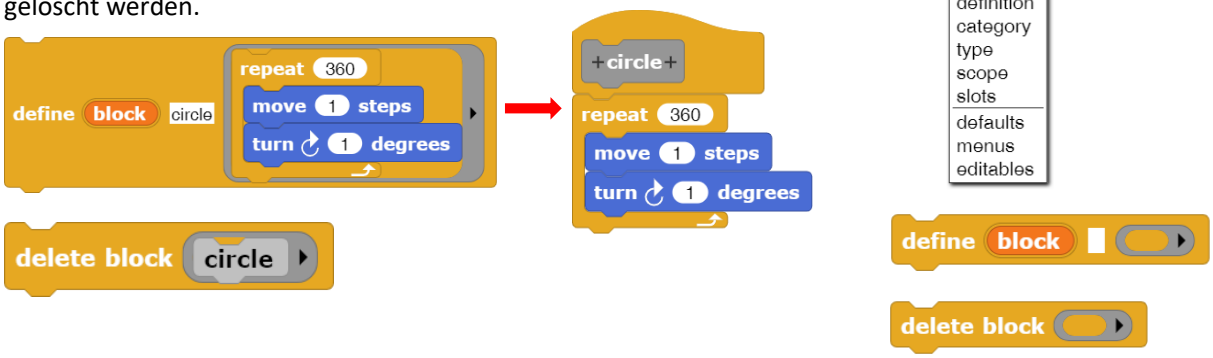
Ein neueres Feature von *Snap!* ist die Möglichkeit der direkten Manipulation von Skripten. Interessieren wir uns z. B. innerhalb eines Skripts für den Inhalt des *get data from drawer <n>*-Blocks, dann erhalten mit *<…> of Block <a block>*-Block das entsprechende Skript. Das können wir mit dem *split*-Block in eine Liste der Teilbefehle umwandeln.



Zu dieser Liste können wir einfach eine Kopie eines Befehls hinzufügen und die Parameter ändern. Die geänderte Liste wird mit *join* zu einem neuen Skript zusammengefügt, das mit dem *call*-Block ausgeführt werden kann.



Das Setzen von Code-Parametern kann während des Programms ebenfalls erfolgen, und der Block selbst kann vollständig erzeugt oder gelöscht werden.



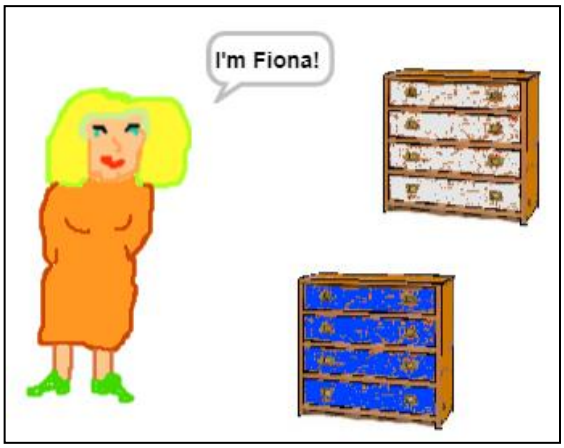
Mithilfe des *clone*-Befehls aus dem Kontextmenüs eines Sprites (s. o.) können wir weitere statische Klone erzeugen. Diese werden zufällig im Ausgabefenster verteilt. Dynamisches Klonen erzeugt ebenfalls neue Sprites, die aber alle an derselben Stelle liegen. Speichert man das Projekt und lädt es neu, dann werden die statisch erzeugten Klone wieder erstellt, die dynamisch erzeugten nicht.[44]

Ein wesentlicher Aspekt der OOP ist die Vererbung. Diese beruht in *Snap!* auf Liebermans[45] Delegation-Modell, das mit Prototypen (also konkreten Objekten, nicht abstrakten Klassen) arbeitet und diese bei Bedarf klont und ggf. verändert. Das Modell wurde weiter vorher beschrieben. Wir werden alle Verfahren zuerst an einfachen Beispielen, danach an komplexeren verdeutlichen.

[44] Das ist ein echter Segen: bei vielen Klonen ist es sonst oft mühsam, die wieder loszuwerden, ohne das Projekt zu zerstören.

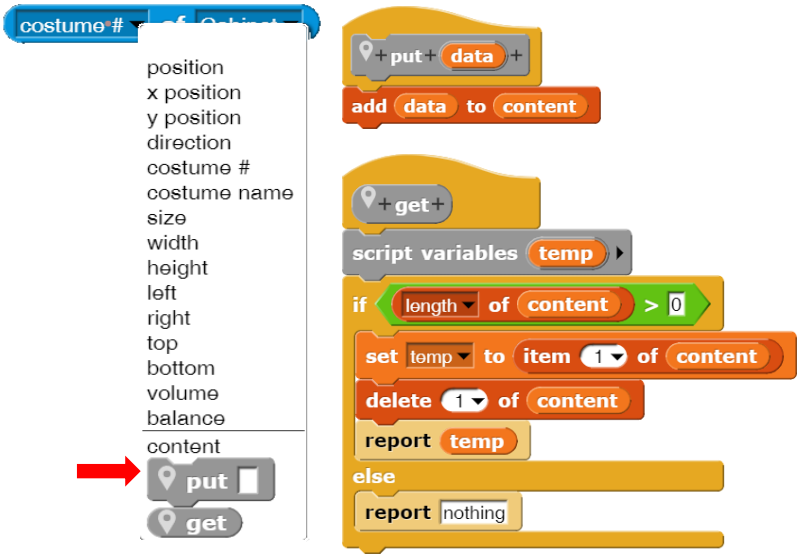
[45] Lieberman, Henry: Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems, ACM SIGPLAN Notices, Volume 21 Issue 11, Nov. 1986

8.1 Fiona und die Aktenschränke



Altersstufe: *Sekundarstufe II*
Material: *Fiona and the filing cabinets*

Wir zeichnen das Kostüm einer eleganten Kommode und erzeugen dafür ein Sprite mit dem Namen *Cabinet*. Die Kommode enthält als Datenspeicher eine lokale Listenvariable *content*, die wir durch eben diese Kommode repräsentieren. Wir statten sie mit lokalen Zugriffsmethoden auf die Daten aus, indem wir die Methoden *put <data>* und *get* implementieren. Das ergibt eine simple *Queue*. Wir können damit beliebige Inhalte in die Liste schreiben und daraus entfernen. Beide Methoden und die Variable werden durch den *<attribute> of <sprite>*-Block angezeigt.

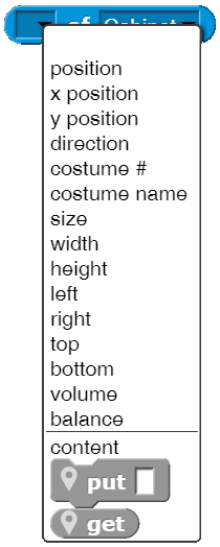


Wir wollen zwei dieser Datenspeicher benutzen. Dazu können wir entweder *Kopien* oder *Klone* der Kommode erzeugen. Bei Kopien werden spätere Änderungen des Prototyps nicht mehr übernommen, bei Klonen aber schon. Eine Ausnahme sind Listenvariable. Hier wird in beiden Fällen eine Referenz auf die Liste kopiert, sodass sich Änderungen an der Liste, z. B. Einfüge-Operationen, auf Klone und Kopien auswirken. Um unabhängige Listen zu erhalten, müssen wir nach dem Klonen diese Verbindung aufbrechen, z. B. indem wir die Liste neu setzen (*set <content> to <list>)* oder kopieren *(set <content> to map <> over <list>)*. Wir entscheiden uns hier für Kopien und erzeugen zwei davon, die Sprites *Papers* und *Souvenirs* mit leicht veränderten Kostümen. Für diese benötigen wir einen Zugriff von außen.

Hilfe erhalten wir von der IT-Beauftragten *Fiona*. Fiona kann die vorhandenen Methoden bei anderen Sprites sehen, aber wie kann sie auf die Datenspeicher zugreifen? Dafür stehen in *Snap!* mehrere Möglichkeiten zur Verfügung, und zwar jeweils für *Commands* und *Reporter*.

Methode eines anderen Sprites finden: 

- Sprite (ggf. Prototyp) im rechten Eingabefeld auswählen:
- Methode im linken Eingabefeld auswählen:

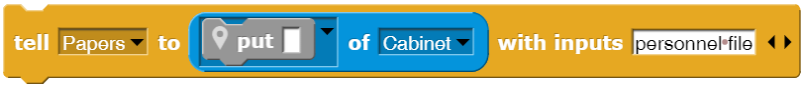


Der Aufruf ergibt den Code des Methodenaufrufs:

lokale Methode eines anderen Sprites ausführen:

Parameter werden in den Feldern nach „with inputs“ der Reihe nach übergeben. Sie werden erst auf Seiten des gerufenen Objekts in die Leerstellen des Methodenkopfs eingefügt, wenn klar ist, welche Methode überhaupt ausgeführt wird.

Commands

mit *tell*:	Anne übermittelt dem angesprochenen Objekt (hier: *Papers*) den auszuführenden Methodenkopf mit den dazugehörigen Parameterwerten (hier: *personnel file*). Das gerufene Objekt folgt auf *tell*. 
mit *run*:	Anne fordert das Objekt *Papers* auf, die übermittelte Methode mit den dazugehörigen Parameterwerten (hier: *personnel file*) auszuführen. Das gerufene Objekt wird im Eingabefenster des *of*-Blocks genannt.  Wichtig: Die Methode wird zuerst ausgewählt, indem ein geeigneter Prototyp oder Klon als Objekt angegeben wird. Erst danach das tatsächlich gemeinte Objekt eingesetzt, das auch z. B. in einer Variablen gespeichert sein kann!
mit *launch*:	wie *run*, nur dass das Skript als eigener Prozess ausgeführt wird, also ohne zu warten.

Reporter	
mit *ask*:	Da es sich um den Aufruf einer Reporter-Methode (einer Funktion) handelt, wird ein Wert zurückgegeben. Eventuelle Parameter werden wie oben geschildert übergeben. Das gerufene Objekt folgt auf *ask*. 
mit *call*:	Vergleichbar mit *run*. Auch hier wird das gerufene Objekt als zweite Eingabe genannt. 

Wenn Attribute eines anderen Sprites von außen geändert werden sollen, dann kann das wie üblich über eine *set*-Methode erreicht werden. Aber es geht auch direkt: wir führen den *set <variable> to <wert>*-Block im richtigen Kontext aus. Dafür muss er in einen grauen Ring verpackt werden, um zu verhindern, dass er schon vor der Ausführung von *run* als Parameter evaluiert wird. Das wäre dann im falschen Kontext. Mit dem Ring wird ein Block als Code übergeben (s. o.), und nicht dessen Ergebnis nach der Ausführung.

Dieser Block ist zu verstehen als: „*Führe den Code, der einer Variablen einen Wert zuweist, im Kontext des Objekts Papers mit den Parameterwerten content und list(1,2,3) aus*“.

Und natürlich können wir auch die Standardblöcke aufrufen.

Fiona als gut ausgebildete IT-Beauftragte kann solche Befehle natürlich absetzen, ein normaler Benutzer aber wohl nicht. Fiona stellt deshalb neue globale Blöcke zur Verfügung, die als Parameter zusätzlich den zu benutzenden Aktenschrank erhalten. Damit wird die Benutzung im gesamten System sehr vereinfacht. Fiona freut sich über die positive Resonanz.

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.

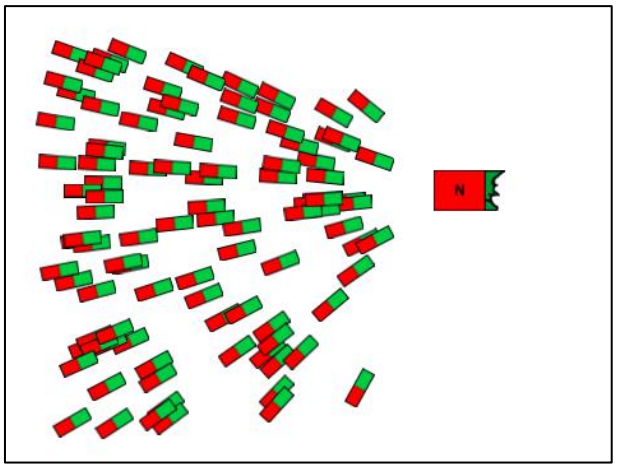
3. Speichern Sie die Daten auf geeignete Weise in **Textdateien**.

4. Organisieren Sie ein „**Datencenter**“, das die Daten einer Firma (einer Schule, einer Familie, …) vorhält, sichert und organisiert. Legen Sie Zugriffsrechte und -methoden fest und implementieren Sie die Verfahren.

8.2 Magnete

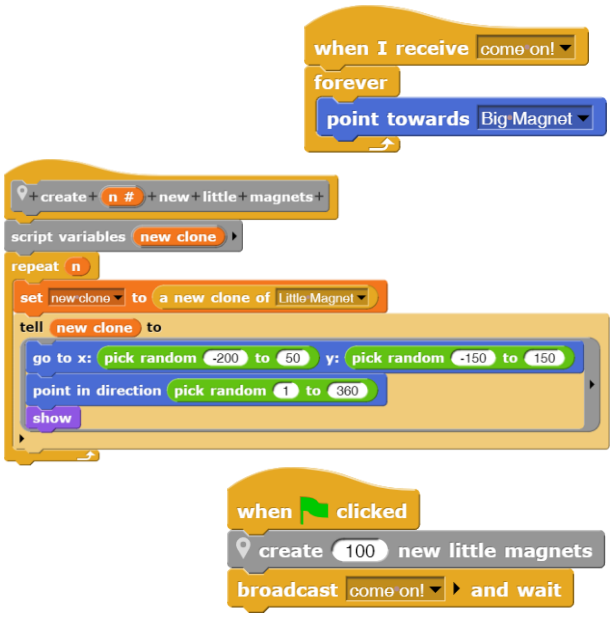
Altersstufe: *Sekundarstufe II* Material: *Magnets*

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. Erhält ein Elementarmagnet die Botschaft „come on!“, dann richtet er sich fortwährend auf den Nordpol aus.



Etwas komplizierter ist das Klonen, weil wir die Klone natürlich im Bildbereich verteilen wollen, etwa so: Wir schreiben die Methode als Block des großen Magneten. In dieser erzeugen wir einen Klon und weisen den einer lokalen Variablen zu. Den Klon schicken wir dann an die Position, die durch die Parameterwerte angegeben wird, drehen ihn in eine beliebige Richtung und lassen ihn erscheinen. Fertig.

Der Umgang mit vielen dynamisch erzeugten Klonen ist extrem einfach: klicken wir den roten Stopp-Button oben-rechts im Fenster an, dann sind alle wieder weg. Und da dynamisch erzeugte Klone nicht im Sprite-Bereich angezeigt werden, sind ihre Skripte richtig schnell. Bewegt man den großen Magneten, dann richten sich alle Elementarmagnete neu aus – sofort.

Aufgabe:

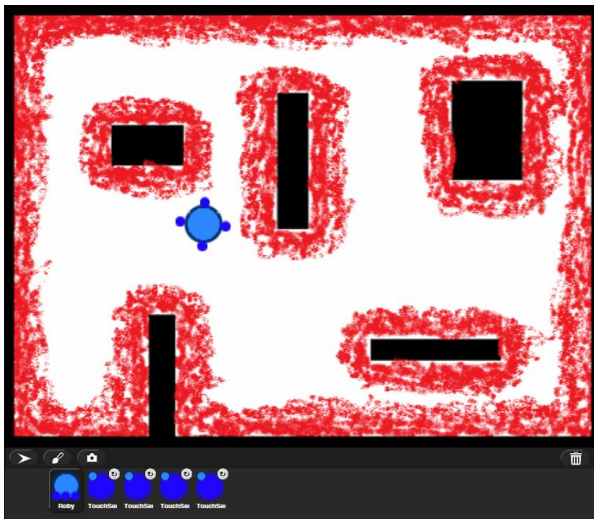
Ergänzen Sie den „Nordpol“ um einen „Südpol“ und bestimmen Sie die Richtung der Kraft auf die Elementarmagnete an deren Orten. Richten Sie Elementarmagnete in diesem Feld aus.

8.3 Ein lernender Roboter[46]

Altersstufe: *Sekundarstufe II* Material: *Learning robot*

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 noch kleineres blaues Sprite, das wir mit einem Prädikat *touching the wall?* ausstatten, also einen Berührungssensor. Dieses Sprite klonen wir dreimal und heften die vier Sensoren dann an den Roboter.[47] Wir nennen sie nach den Himmelsrichtungen *TouchSensorN*, *TouchSensorE*, … usw. Es entsteht eine *Aggregation*. Den Roboter statten wir mit zwei lokalen Variablen *vx* und *vy* aus, die seine 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:

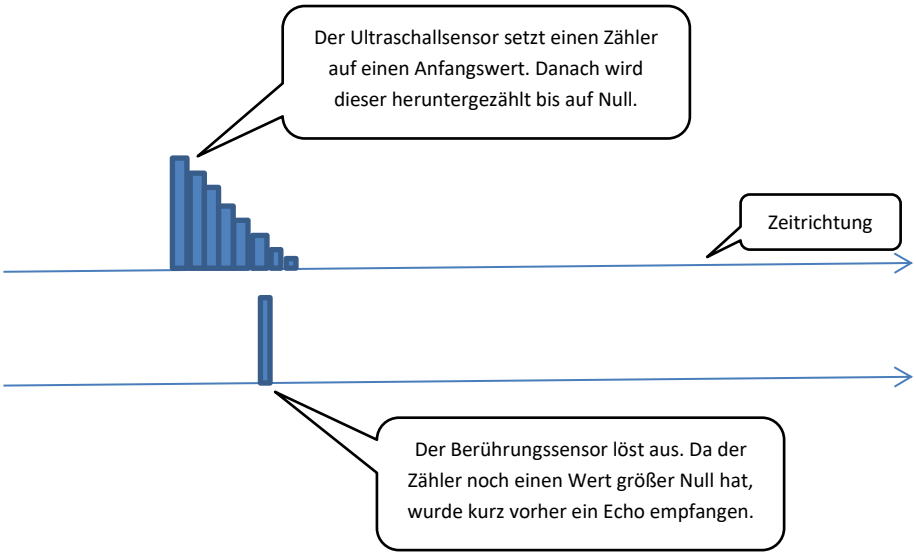


[46] Das Beispiel hat den Laufroboter von Prof. Florentin Wörgötter, Bernstein Zentrum für Computational Neuroscience Göttingen, als Vorlage, beschrieben z. B. in http://www.chip.de/news/Schnellster-Roboter-lernt-berg-auf-zu-gehen_27892038.html

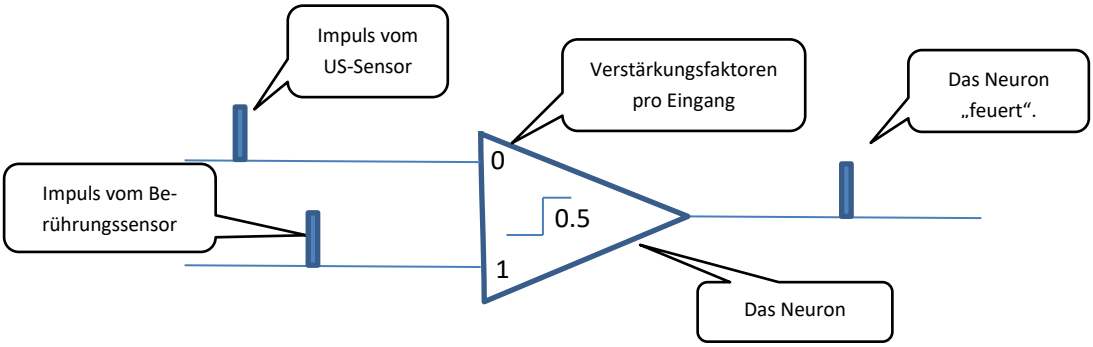
[47] Im nächsten Kapitel wird beschrieben, wie man Aggregationen von Sprites erzeugt, also Sprites an andere heftet.

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. Wir nennen sie *USsenorN*, …

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: 100) 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.

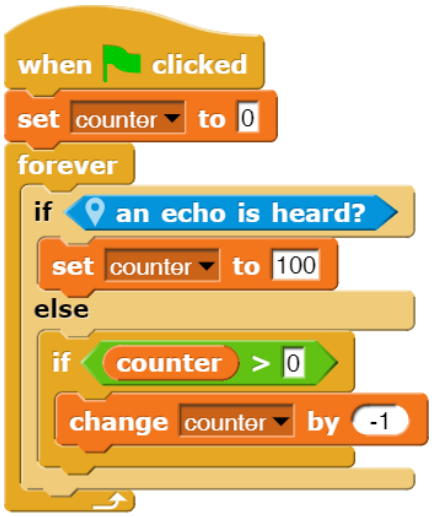


Durch diese Konstellation wird ein Lernschritt in Gang gesetzt, der in einem *Neuron* stattfindet. Dieses verfügt über zwei Eingänge, die vom zugeordneten Berührungssensor bzw. Ultraschallsensor kommen und jeweils mit einem Gewicht (*weight*) behaftet sind, sowie einen Schwellwert. Die Leitung vom Berührungssensor 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 *0*. 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.

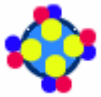


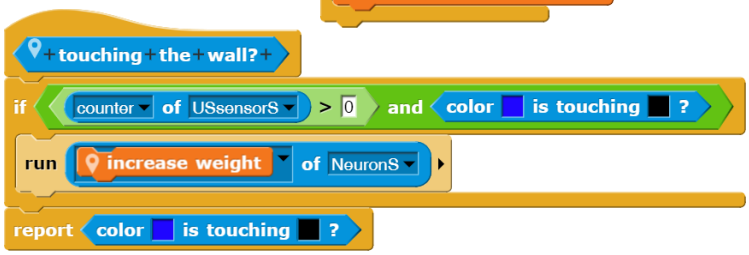
Diese Form *Pawlowschen Lernens* realisieren wir jetzt.

Der Ultraschallsensor arbeitet genau wie oben beschrieben. Der Zugriff auf das lokale Attribut *counter* kann direkt mit dem *<attribute> of <object>*-Block erfolgen. Die eigentlichen Änderungen finden also in den Berührungssensoren und den vier zugeordneten Neuronen statt. Da diese Klone des jeweils einzigen Prototyps sind, genügt es fast, nur in diesem die Ergänzungen vorzunehmen. Die Klone übernehmen sie, weil sie die Methoden des Prototyps erben. Allerdings müssen wir danach noch angeben, auf welches Element der vier Gruppen das Sprite reagieren soll.



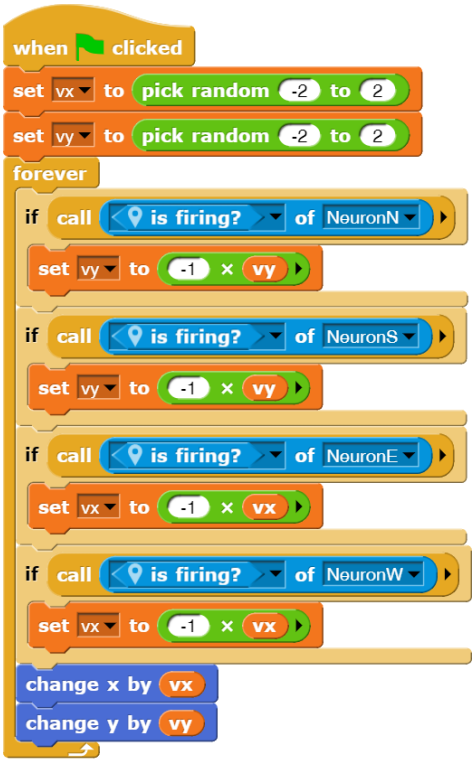
Beim Berühren einer Wand muss noch festgestellt werden, ob der zugehörige Ultraschallsensor „kurz vorher“ ausgelöst hat. In den Klonen überschreiben wir danach die geerbte „blasse“ Methode, indem wir den zugeordneten Sensor verstellen. Damit verschwindet auch die Blässe. Vorher haben wir den Ultraschallsensor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



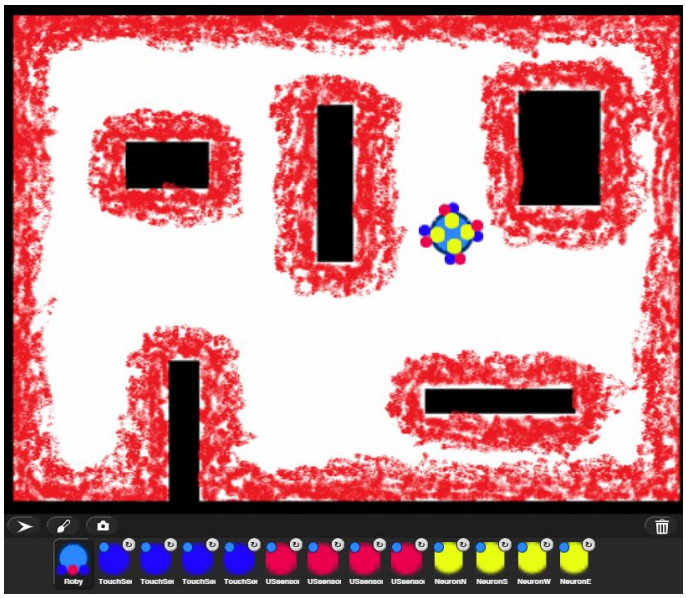


Das Neuron benötigen noch ein Prädikat *is firing?*, das so wie oben beschrieben arbeitet.

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“. Er ist richtig schlau geworden! 😉

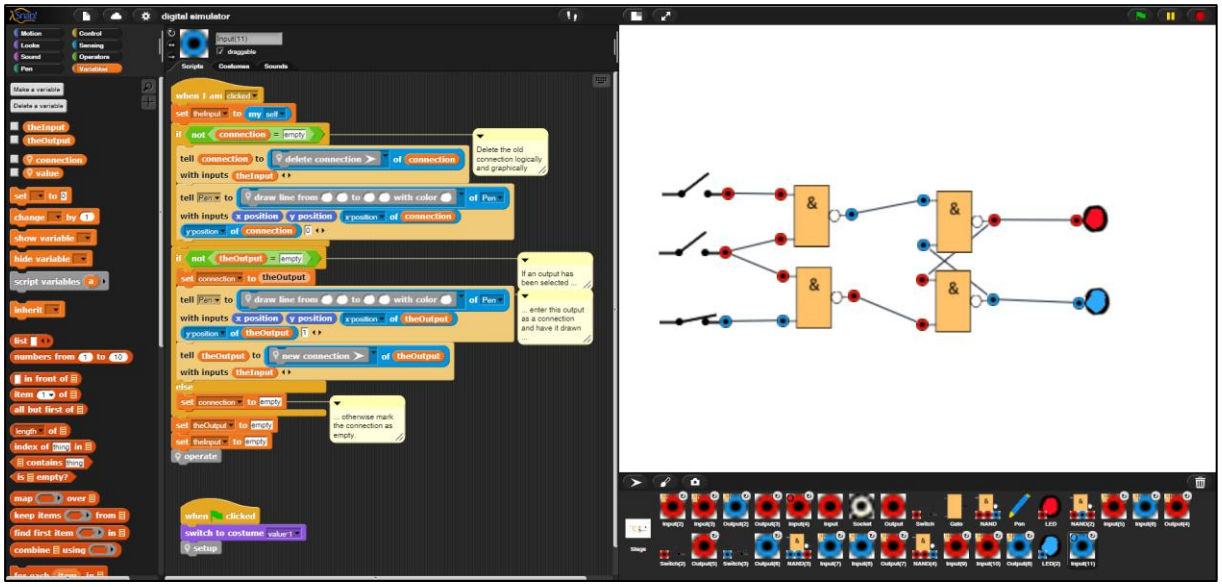


Aufgaben

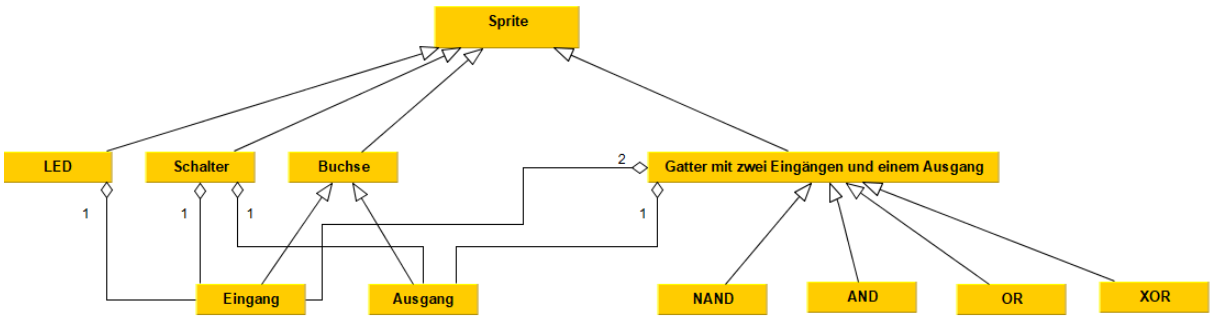
1. Verpassen Sie dem Programm eine **Oberfläche**, mit der sich die wesentlichen Faktoren ändern leicht lassen: die Geschwindigkeit, die Gewichte, die Schwellwerte.
2. Führen Sie weitere **Sensortypen** sowie weitere Ereignisse neben den Kollisionen ein.
   a: Lassen Sie Roby Korrelationen zwischen Sensorwerten und Ereignissen in unterschiedlichen „Welten“ finden. Roby passt sich so seiner Umgebung an.
   b: Diskutieren Sie weitere Möglichkeiten, dass Roby sich an eine veränderliche Umwelt anpasst.
3. Diskutieren Sie den Bedarf nach „**Vergessen**“ sowie Möglichkeiten, diesen Prozess zu realisieren.
4. Ersetzen Sie Roby durch eine Maus mit einem Käsesensor. Setzen Sie die in ein **Labyrinth**. Dort soll sie den Käse suchen.

8.4 Ein Digitalsimulator

Altersstufe: *Sekundarstufe II* Material: *Digital simulator*

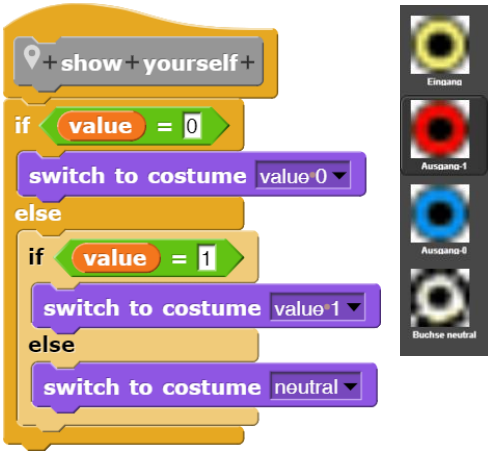


Ein *Digitalsimulator* ist ein Programm, mit dem sich digitale Schaltungen simulieren lassen. Es besteht aus Schaltern, LEDs und Gattern, in diesem Fall nur aus NANDs (Not AND), aus denen sich alle anderen Schaltungen aufbauen lassen. Auf den Bauteilen sitzen unterschiedliche Arten von Buchsen, mit deren Hilfe Signale weitergeleitet werden. Wir können die Zusammenhänge übersichtlich in einem (vereinfachten) UML-Diagramm darstellen. Die Vererbung erfolgt in diesem Fall über Delegation.



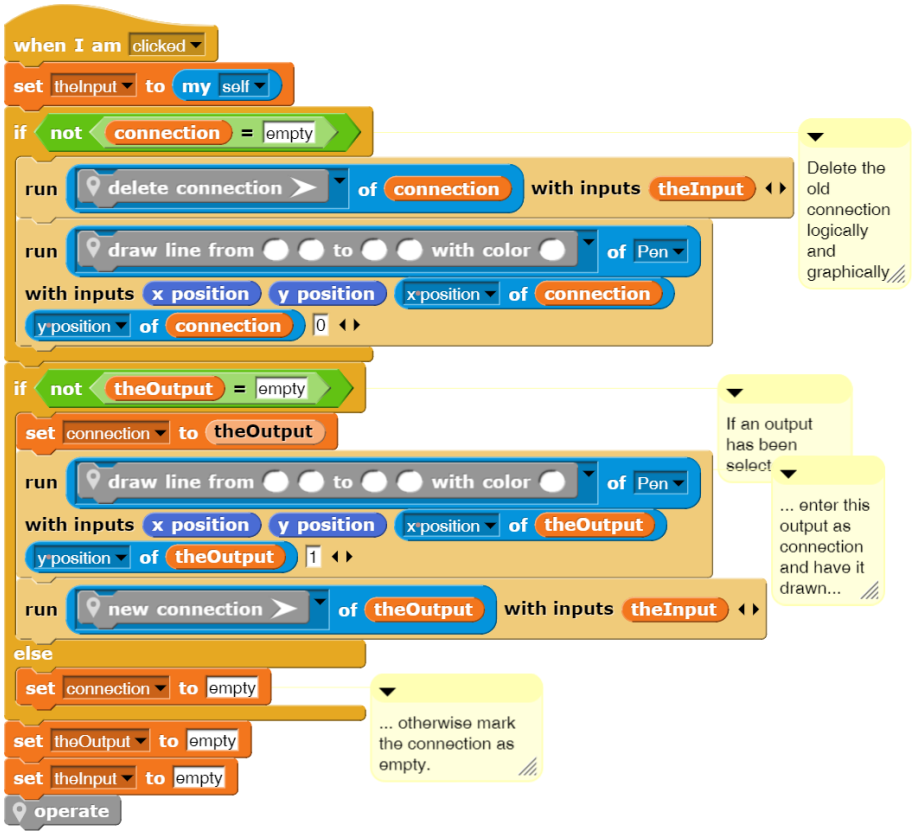
Buchsen und Verbindungen

Als „Mutter aller Buchsen“ zeichnen wir eine *neutrale Buchse (socket)*, die als Prototyp für Eingangs- (*Input*) und Ausgangsbuchsen (*Output*) dient. Alle Buchsen haben einen Wert (*value*), der *0* oder *1* sein kann. Eingänge erhalten ihren Wert vom angeschlossenen Kabel oder sie erhalten aus technischen Gründen den Wert *1*, falls sie nicht verbunden sind. Ausgänge erhalten ihren Wert vom Bauteil, auf dem sie sitzen. Sie stellen also das Ergebnis einer logischen Schaltung dar. Alle Buchsen erben von der neutralen Buchse die Methode *show yourself*, die ihren Wert farblich darstellt, sowie die lokale Variable *value*.

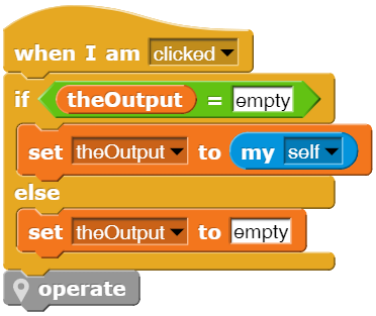


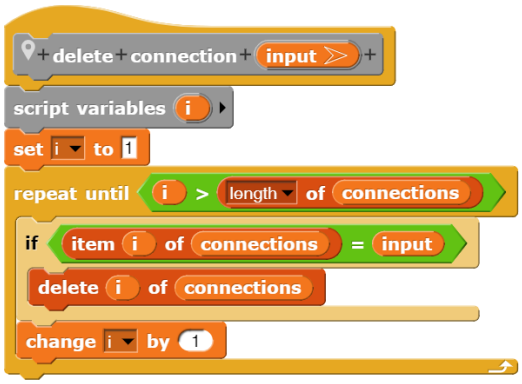
Wir erzeugen mithilfe des Kontextmenüs (*clone*) zwei Klone der neutralen Buchse, die uns im Weiteren als Prototypen für Ein- und Ausgänge dienen.

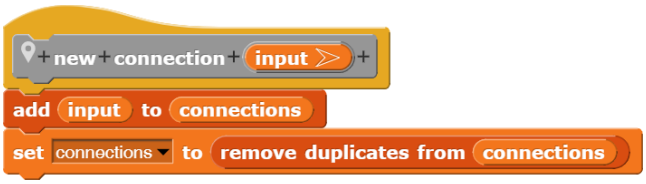
Buchsen sollen verbunden werden, indem wir zuerst einen Ausgang anklicken und danach einen Eingang. Wird nur der Eingang angeklickt, dann wird dessen Verbindung zu einem Ausgang gelöscht – wenn sie denn existiert. Dargestellt werden Verbindungen nur rudimentär als Linien auf der Bühne. Verschiebt man danach die Schaltelemente, dann bleiben die Linien „frei im Raum“ stehen.[48] Eingänge können höchstens mit einem Ausgang verbunden sein. Dafür erhalten sie eine zusätzliche Variable *connection*. Ausgänge können ihre Werte an mehrere Eingänge verteilen, deshalb erhalten sie eine Listenvariable *connections*, in die die verbundenen Eingänge ein- oder ausgetragen werden. Wird ein Ausgang angeklickt, dann erhält die globale Variable *theOutput* diesen Ausgang als Wert. Wird ein Eingang angeklickt, dann sorgt er für die Aktualisierung der Verbindungen.



Ausgänge haben es etwas einfacher: sie stellen die Möglichkeiten zum Ein- und Austragen von Verbindungen bereit – und warten, was kommt.

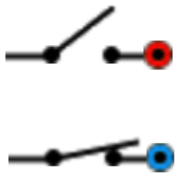




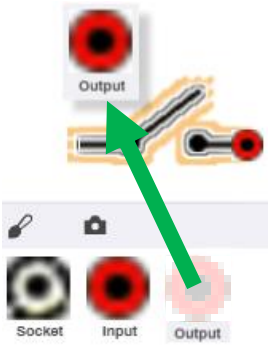


[48] Die Darstellung und vor allem die Verteilung von Leitungen ist ein eigenständiges Problem.

Schalter



Schalter dienen dazu, Eingangswerte zu verändern. Wir erzeugen zwei Kostüme, die den offenen bzw. geschlossenen Zustand darstellen. Das linke Ende des Schalters lassen wir offen, es symbolisiert die Verbindung zur Erde und hat so den festen Wert 0. Ans rechte Ende kommt eine Ausgangsbuchse, die entweder den Wert *1* (Zustand „offen“) oder *0* (Zustand „geschlossen“) erhält. Die neue Buchse erhalten wir durch Klonen der Ausgangsbuchse. Danach schieben wir das erhaltene Sprite an die richtige Stelle beim Schalter.

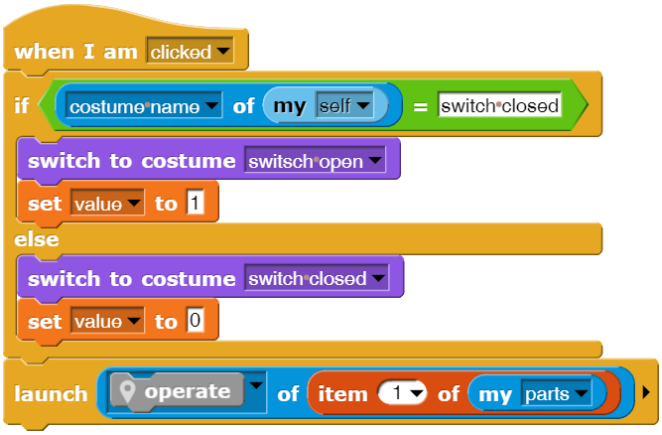


Dort muss es jetzt verankert werden. Dazu schieben wir das Sprite-Symbol des Ausgangs aus dem Sprite-Bereich über den Schalter im Ausgabefenster. Dessen Umriss leuchtet auf, wenn er merkt, dass er gemeint ist. Damit ist die Buchse am Schalter befestigt: er ist der *Anker* der entstehenden *Aggregation*.

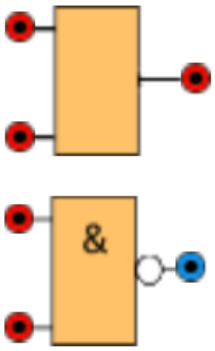
Eine Aggregation von Sprites wird also erzeugt, indem die Elemente auf der Bühne zuerst an die richtigen Stellen gezogen werden und danach die Sprite-Symbole aus dem Sprite-Coral auf das Anker-Element gezogen werden. Die angefügten Sprites (hier: die Buchse) werden Elemente der *parts*-Liste des Ankers (hier: des Schalters) und werden am Sprite-Symbol des Ankers angezeigt. Mit *detach from …* aus dem Kontextmenü der angefügten Sprites können sie wieder vom Anker gelöst werden.

Da wir die Bauteile unseres Digitalsimulators per Maus bedienen wollen, bietet es sich an, dass der Schalter auf Mausklicks reagiert. Das ist einfach zu erreichen: bei jedem Klick wechselt er das Kostüm. Dazu muss er wissen, wie er gerade aussieht: mit *<costume-name> of <my self>* erhält er das aktuelle Kostüm.

Wir benötigen noch einen Mechanismus, um die Reaktionen der *parts* zu steuern, in diesem Fall der Ausgangsbuchse. Da es übertragbar sein soll, muss das Verfahren allgemein brauchbar sein. Wir statten also jedes Bauteil mit einer *operate*-Methode und einer Variablen *value* aus. Ändert sich der Zustand des Schalters, dann ändert der seinen Wert. Zuletzt ruft er die *operate*-Methode des Ausgangs auf – der ist hier das einzige Element seiner *parts*-Liste. Wir nutzen den *launch*-Block, um die Programmausführung nicht warten zu lassen.



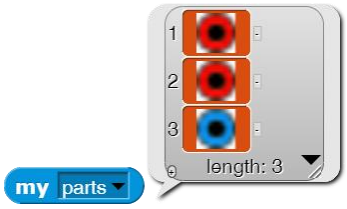
Gatter

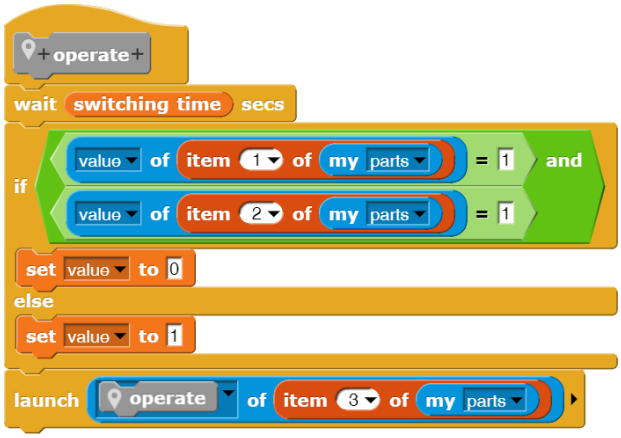


Zum Erzeugen von Gattern führen wir zuerst mal einen Prototyp *Gate* ein, der über zwei Eingänge und einen Ausgang verfügt. Weiterhin enthält er eine Variable *switching time*. Die erforderlichen Buchsen fügen wir wie bei den Schaltern gelernt an. Aus diesem Gatter lassen sich dann andere Gatter wie AND, OR, XOR oder NAND ableiten. Für das NAND erzeugen wir einen Klon des Gatters namens *NAND* und versehen ihn mit einem angepassten Kostüm.

Die aus dem Gatter abgeleiteten Prototypen erben die *operate*-Methode des Gatters und die Instanzvariable *value*. Beide sind natürlich eigentlich überflüssig, da das Gatter gar keine richtige Funktion hat. Wir lassen die Methode deshalb leer und überschreiben sie bei den abgeleiteten Prototypen. (Sollten wir etwas vergessen haben, dann können wir auch nachträglich Variable und Methoden im Prototyp erzeugen. Diese werden an die Klone sofort vererbt.) Geerbte Größen erscheinen bei Klonen etwas heller als eigene. Werden sie überschrieben, dann erhalten die geänderten Größen die normale Farbe.

Die *operate*-Methode des NANDs ist einfach zu schreiben. Der *my <parts>*-Block zeigt uns die Ein- und Ausgänge des NANDs an. Deren Werte können wir auslesen bzw. so wie beim Schalter setzen. Wir nutzen wieder den *launch*- statt des *run*-Blocks.



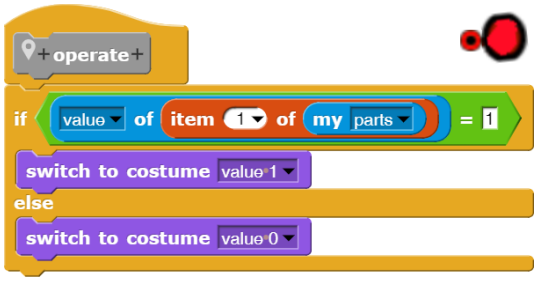


Der Stift

Der Stift stellt nur eine einfache Methode bereit, mit deren Hilfe gerade Linien in unterschiedlichen Farben auf der Bühne gezeichnet werden können. Weitere Aufgaben hat er nicht.

LEDs

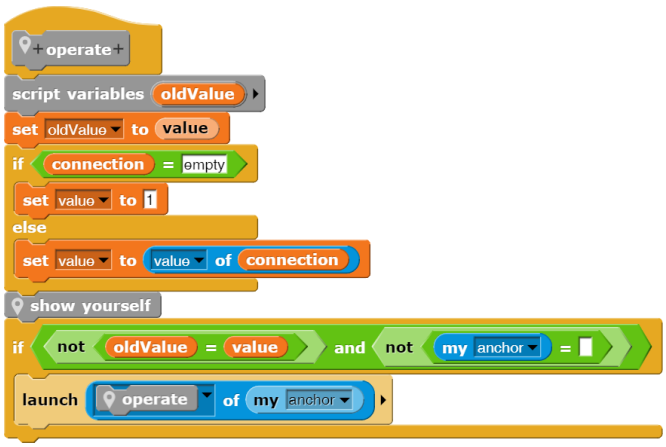
Als ein sehr einfaches Beispiel für das Hinzufügen neuer Bauteile zum System führen wir den Prototyp einer *LED* (Leuchtdiode) ein. Diese erhält zwei Kostüme für die Werte *0* und *1* sowie einen Eingang. Da dieser sich gut im System auskennt, kann sich die LED voll auf ihn verlassen und sich auf das beschränken, was LEDs so machen – leuchten. Mehr ist nicht zu tun.

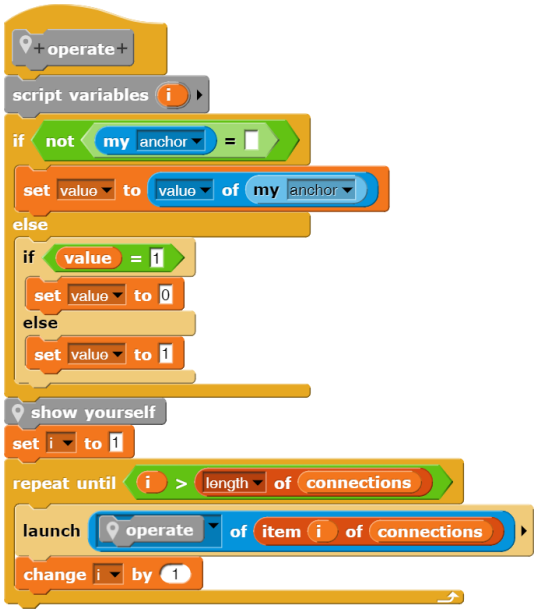


Das Zusammenspiel der Komponenten

Die Aktivität soll unsere Schaltnetz wellenartig in einer Art *feed-forward*-Verfahren durchlaufen: Jedes Bauteil benachrichtigt die verbundenen Teile und ruft deren *operate*-Methode auf, wenn sich etwas geändert hat. Sitzt also z. B. eine Ausgangsbuchse auf einem Schalter, dann ruft der die *operate*-Methode des Ausgangs auf, wenn er angeklickt wurde und deshalb seinen Wert änderte. Der wiederum aktiviert alle verbundenen Eingänge. Jeder dieser Eingänge ruft die *operate* -Methode des Gatters auf, auf dem er sitzt – allerdings nur, wenn sich sein Wert geändert hat. Falls nicht, wird die Welle hier also gestoppt. Bisher kann es sich beim Gatter nur um ein NAND handeln. Dieses wartet seine Schaltzeit ab, liest die Werte seine Eingänge und aktiviert den Ausgang – usw.

Als Beispiel dienen die *operate* -Methoden von Eingang und Ausgang.





Aufgaben

1. Erzeugen Sie nach dem Vorbild des NANDs Prototypen für folgende **Gatter**:
   a: ein UND (AND) b: ein ODER (OR)
   c: ein Exklusiv-Oder (XOR) d: ein Nicht-ODER (NOR)

2. Erzeugen Sie einen Prototyp für **NICHT**-Gatter (NOT), die nur einen Eingang und einen Ausgang besitzen.

3. Erzeugen Sie einen Prototyp für einen **Taktgeber**. Die Taktfrequenz soll regelbar sein.

4. Erzeugen Sie einen Prototyp für **RS-FlipFlops** (RS-FF). Informieren Sie sich zuvor über deren Arbeitsweise.

5. Erzeugen Sie einen Prototyp für **JK-Master-Slave-FlipFlops** (JK-FF). Informieren Sie sich zuvor über deren Arbeitsweise.

6. Unsere Gatter reagieren erst nach einer **Schaltzeit**, die unterschiedlich sein kann. Warum eigentlich?

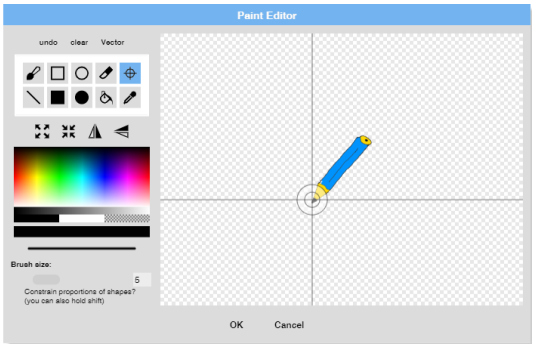
7. Entwickeln Sie eine **Oberfläche** mit Buttons, Auswahlboxen, … für den Digitalsimulator, mit deren Hilfe Bauelemente erzeugt und gelöscht werden können, Elemente ausgewählt werden, die Simulation gestartet und wieder gestoppt werden kann, ...

9 Grafik

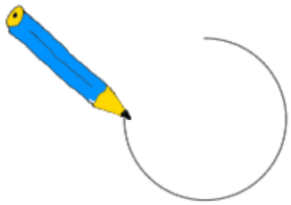
9.1 Liniengrafik mit Koch- und Hilbert-Kurve

Altersstufe: *Sekundarstufe I/II* Materialien: *Snowflake, Hilbert curve*

In *Snap!* verfügt jedes Sprite über einen (virtuellen) Stift, mit dem es auf der Bühne zeichnen kann. Die Funktionalität entspricht der bekannten *Turtle-Grafik*[49]. Die Blöcke dafür finden sich in den beiden Paletten *Pen* und *Motion*. In der ersten wird der Stift gesteuert, d. h. angehoben oder abgesenkt, Stiftfarbe und -breite eingestellt, … In der zweiten finden sich die Befehle zum Bewegen des Sprites. Bei dieser Bewegung hinterlässt der Stift je nach Zustand „Spuren“, die dann die erzeugte Liniengrafik bilden – und die als *pen trails* auch weiter verarbeitbar sind. Zu beachten ist, dass sich der Stift im *rotation center* des aktuellen Kostüms des Sprites befindet. Man kann dieses im Kostüm-Editor mithilfe des Fadenkreuz-Werkzeugs verschieben.



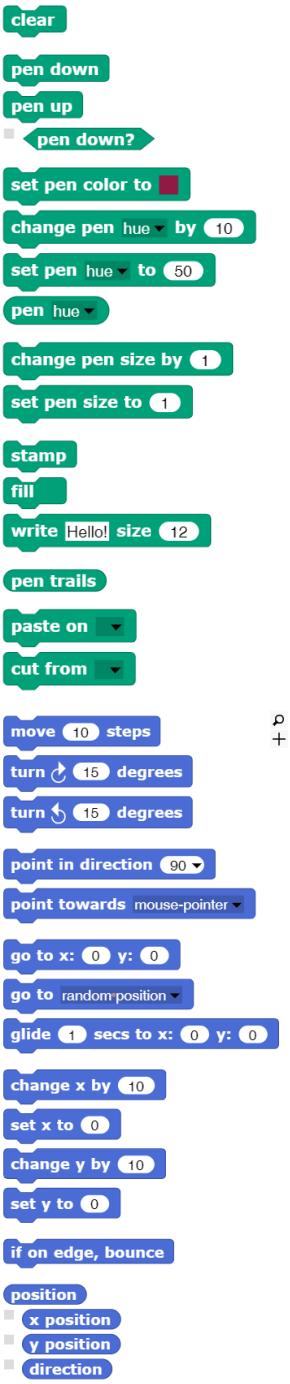
Wählen wir als Kostüm den schon bekannten *Pen*, dann erzeugt das nebenstehende Skript einen einfachen Kreis.



```
go to x: 0 y: 0
clear
pen down
repeat 360
    move 1 steps
    turn 1 degrees
pen up
```

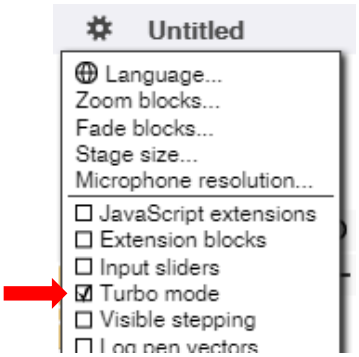
Anhand des Beispiels lässt sich gut die Wirkung des *Warp*-Blocks demonstrieren. Während ohne diesen der Stift recht gemütlich den Kreis zeichnet, erscheint der fertige Kreis mit *Warp*-Block praktisch sofort. Der Grund liegt darin, dass im ersten Fall nach jeder Blockausführung der Zustand des Systems neu gezeigt wird, während das im zweiten Fall nur in größeren Abständen erfolgt. Der Unterschied ist „dramatisch“.

```
warp
    go to x: 0 y: 0
    clear
    pen down
    repeat 360
        move 1 steps
        turn 1 degrees
    pen up
```



[49] https://de.wikipedia.org/wiki/Turtle-Grafik

Eine ähnliche Beschleunigung lässt sich mithilfe der *Turbo mode* -Option im Einstellungsmenü erreichen. Dieser bezieht sich aber auf die gesamte Programmausführung und nicht nur auf einen ausgewählten Bereich.

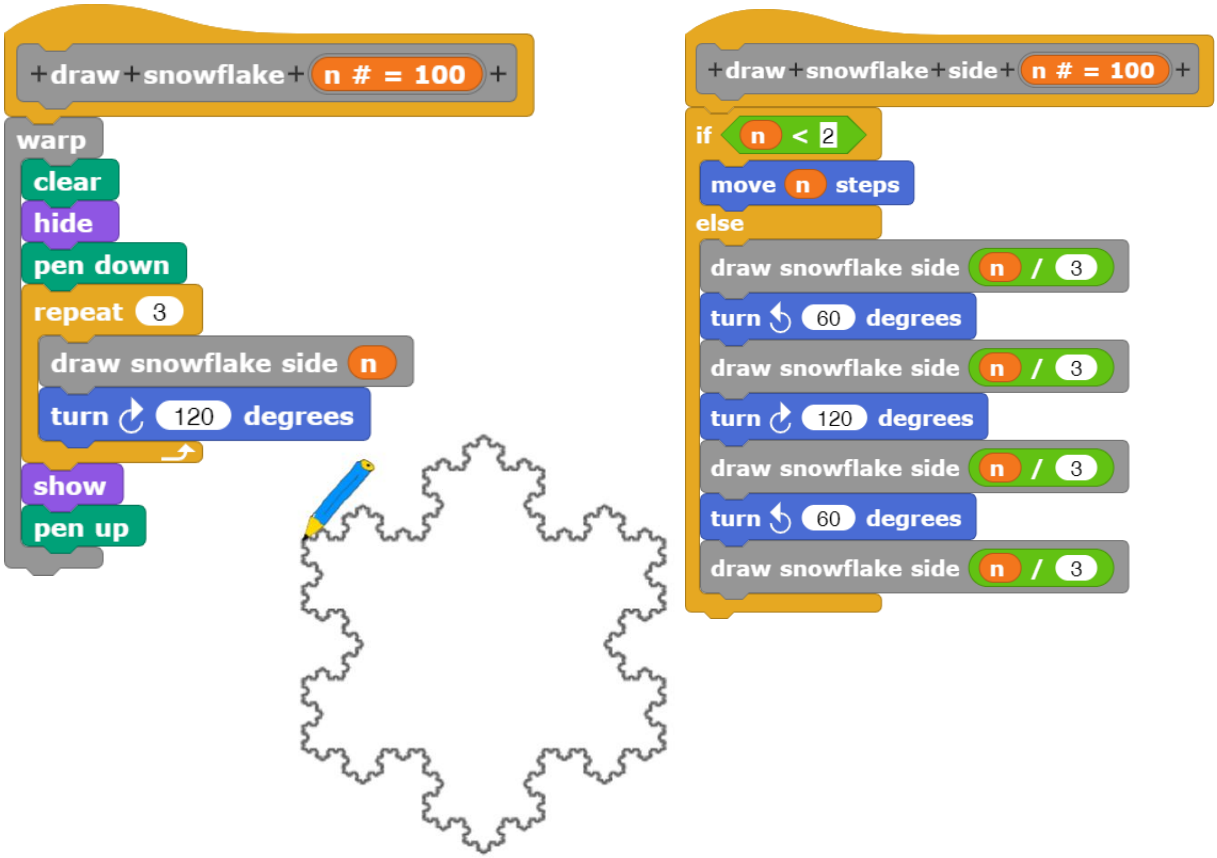


Mithilfe der Turtlegrafik lassen sich einige der bekannten rekursiven Kurven sehr elegant zeichnen. Wir beginnen mit der *Schneeflocken-* (oder *Koch-*) *Kurve*. Sie entsteht, indem auf den Seiten eines Dreiecks immer wieder in der Mitte Dreiecke „ausgestülpt" werden, solange bis die Seiten zu kurz für diesen Prozess werden. In diesem Fall werden die Seiten nur als gerade Linien gezeichnet. Eine Schneeflocke entsteht, indem ein gleichseitiges „Dreieck" aus drei solchen Seiten zusammengesetzt wird.

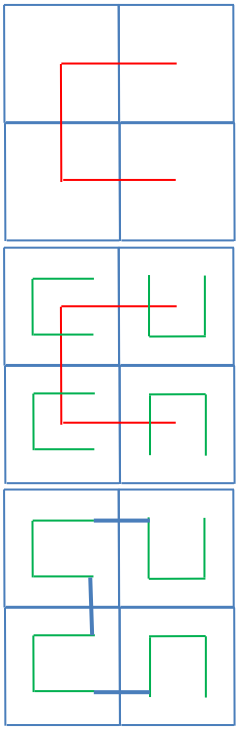
zeichne Schneeflockenseite der Länge n

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

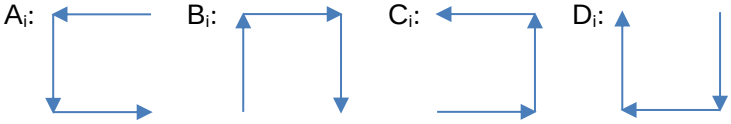
Das Verfahren lässt sich direkt nach *Snap!* übersetzen:



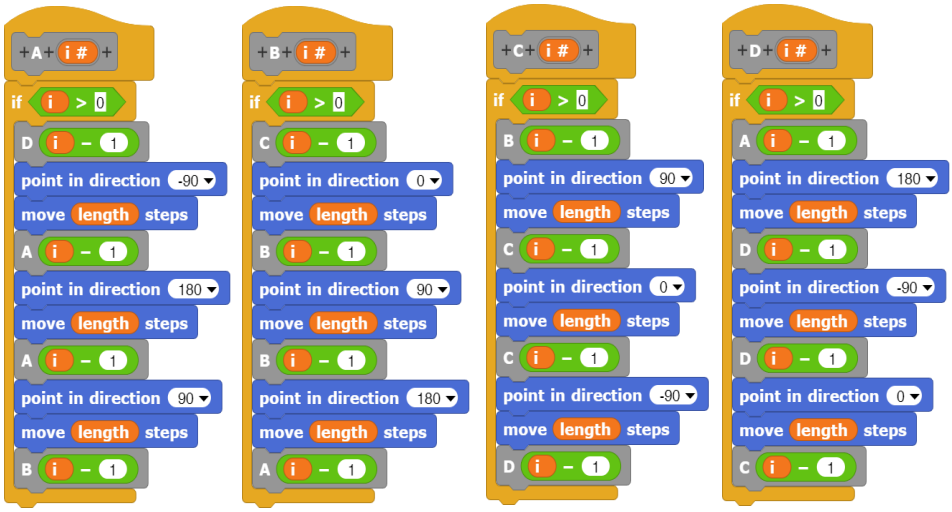
Zur Konstruktion der *Hilbertkurve* verwenden wir eine Version nach László Böszörményi[50]. Es handelt sich um eine der flächenfüllenden Kurven, die als Generator eine Art Kasten hat. Die Ecken des Kastens liegen in den Mittelpunkten der vier Quadranten eines Quadrats. In der nächsten Stufe wird dieser 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.

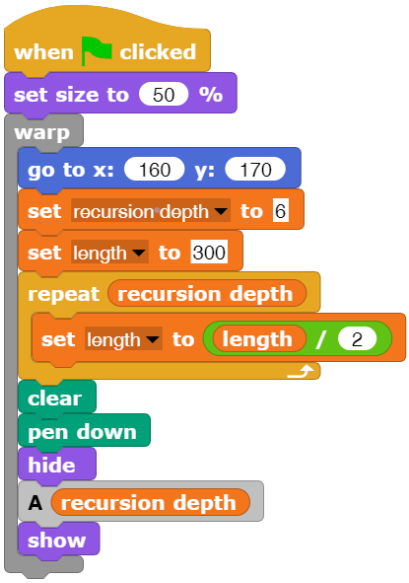


In der Version von Böszörményi werden die Kästen je nach Orientierung und Umlaufrichtung mit A bis D gekennzeichnet.

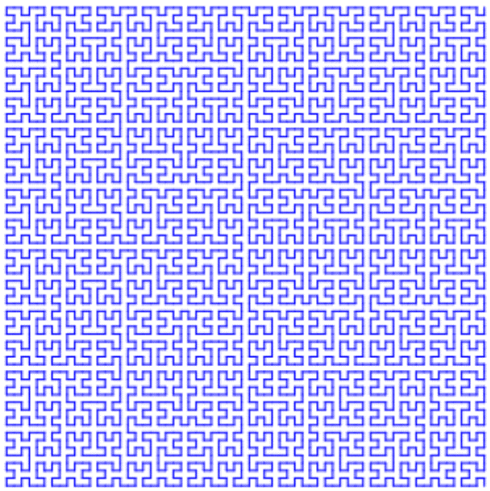


Aus diesen Elementen wird die Hilbertkurve zusammengesetzt, indem man mit A beginnt und die anderen Elemente „verdreht“ aufruft. Der Parameter *i* gibt die Rekursionstiefe und damit die Größe der Elemente an. Er wird „heruntergezählt“ bis auf Null.





Der Aufruf erfolgt wie beschrieben, nachdem das Sprite zum Anfangspunkt rechtsoben geschickt wurde. Die endgültige Länge der zu zeichnenden Teilstrecken wird aus der Rekursionstiefe ermittelt – und dann wird gezeichnet. Auch hier ist die Wirkung des *Warp*-Blocks drastisch.

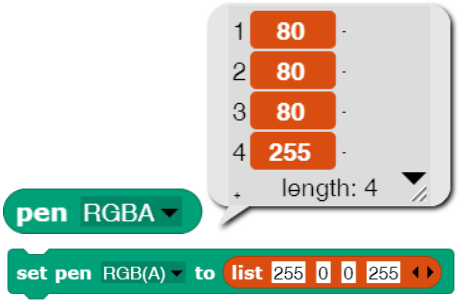


[50] http://bscwpub-itec.uni-klu.ac.at/pub/bscw.cgi/d11952/10.%20Rekursive%20Algorithmen.pdf

9.2 Der RGB-Farbwürfel

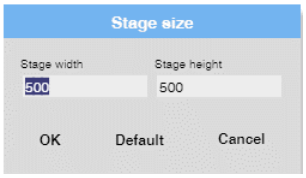
Altersstufe: *Sekundarstufe I* Material: *Color cube on stage*

Mutig geworden durch diesen Erfolg versuchen wir als Nächstes, den bekannten Farbwürfel des RGB-Farbraums[51] selbst zu erzeugen. Dazu müssen wir natürlich die RGB-Farben für den Pen einstellen können. Möglichkeiten dafür finden wir in der Pen-Palette. Zuerst sehen wir uns mal an, wie diese RGB(A)-Farben dargestellt. Das kennen wir doch schon!

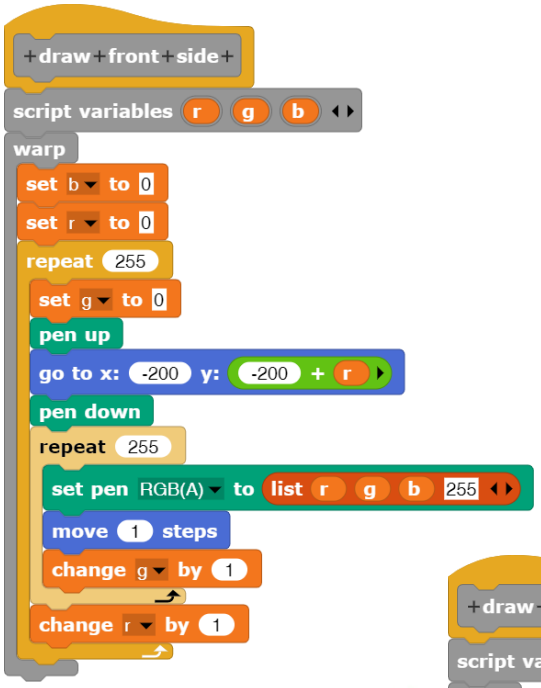


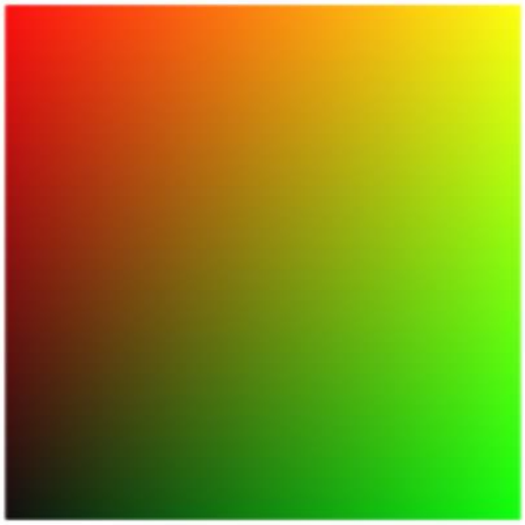
Auf die gleiche Art können wir dann die Pen-Farbe auch setzen.

Na denn: Wir vergrößern die Bühne auf die Maße 500 x 500 Pixel, indem wir die entsprechenden Einträge im Settings-Menü ändern. Dann wird gezeichnet.

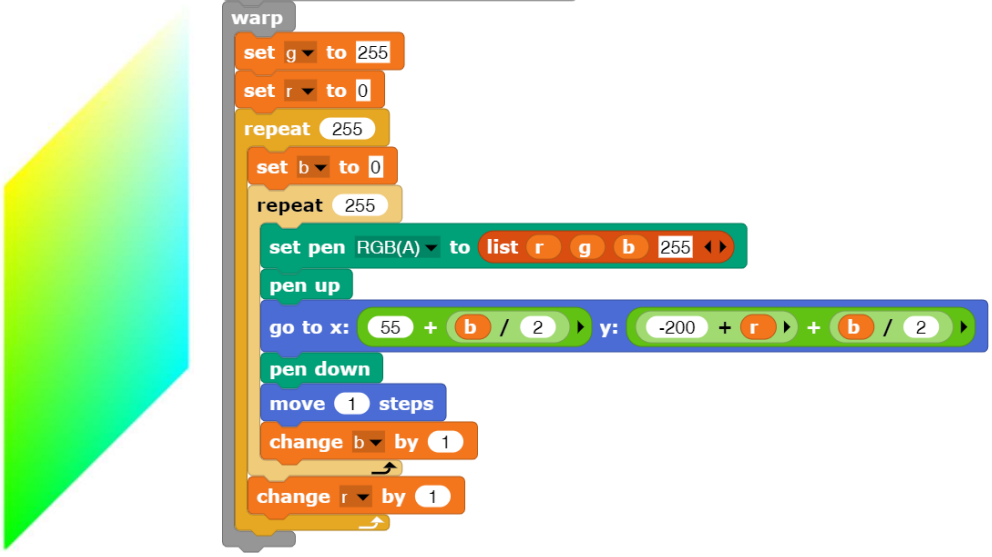


Zuerst einmal die Vorderseite des Farbwürfels.



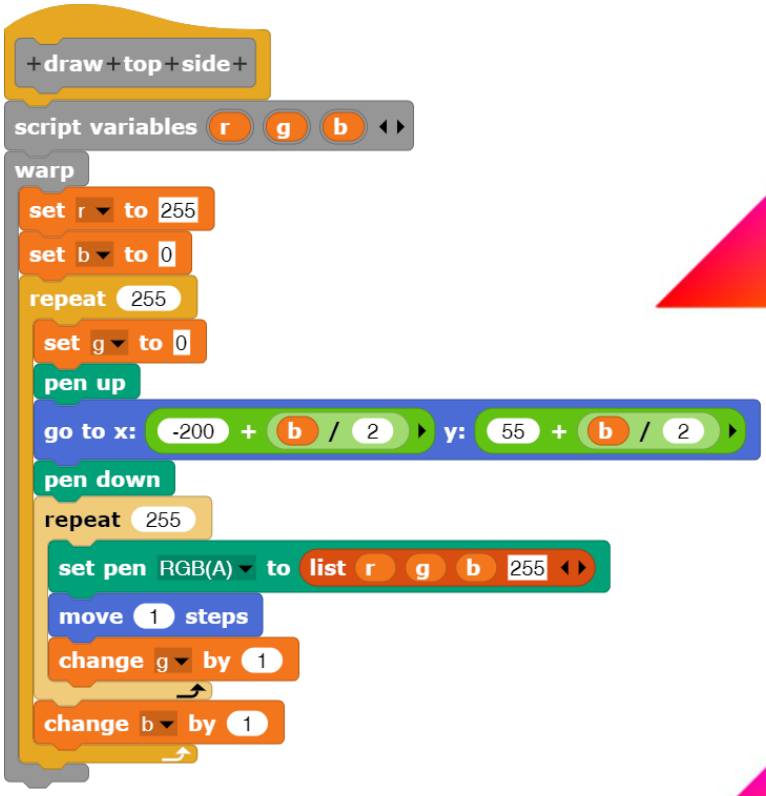


Dann die rechte Seite.

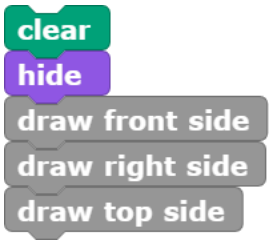


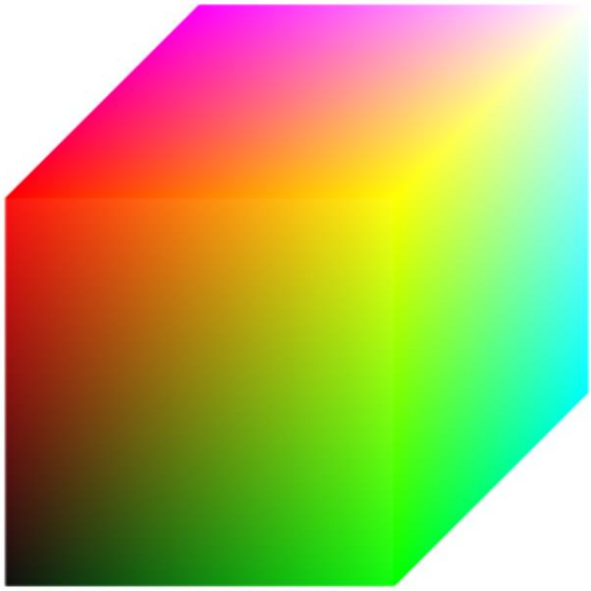
[51] https://de.wikipedia.org/wiki/RGB-Farbraum

Und schließlich den Deckel drauf.

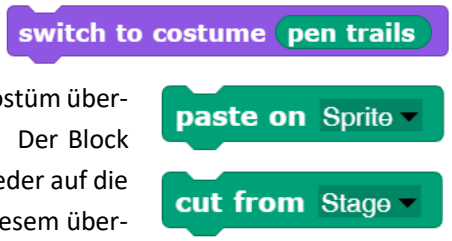


Das ergibt insgesamt den RGB-Farbraum.



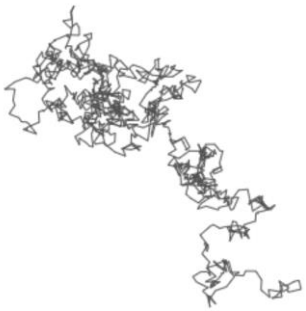


9.3 Bedrucken und Beschneiden von Kostümen

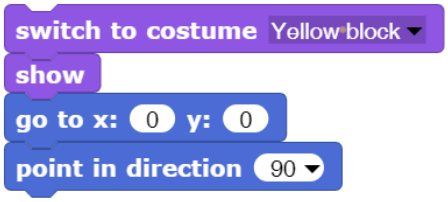


Die Stifte der Sprites zeichnen auf der *Stage*, Pixelgrafik ist dagegen auch auf Kostümen von Sprites möglich. Mithilfe des *pen trails*-Blocks lässt sich der momentane Zustand der Bühne in ein Kostüm überführen, das sich ggf. auch wieder auf die Bühne „drucken“ lässt. Der Block *paste on <target>* druckt das aktuelle Kostüm eines Sprites entweder auf die Bühne oder ein ausgewähltes anderes Sprite, soweit es sich mit diesem überschneidet. Der Block *cut from <target>* schneidet den Bereich des eigenen Kostüms aus dem Kostüm des angegebenen Sprites heraus.

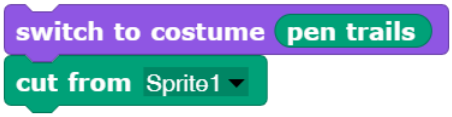
Als Beispiel erzeugen wir zuerst ein „Gekrakel“ auf der Bühne.



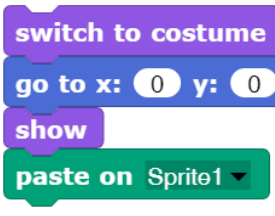
Dann verpassen wir dem Sprite ein gelbes Rechteck als Kostüm und schicken es wieder in die Mitte.

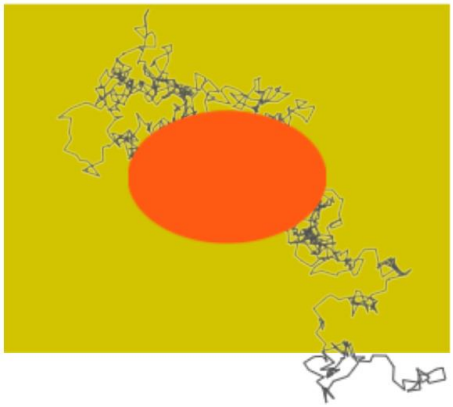


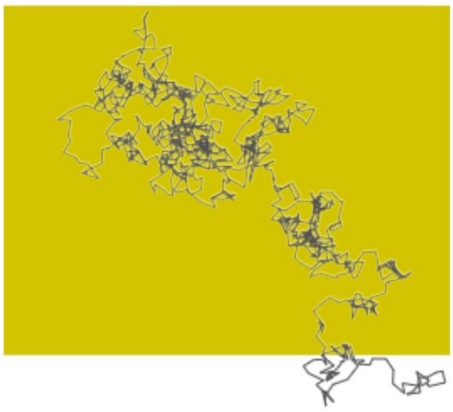
Wir erzeugen ein zweites Sprite und geben dem das Kostüm der *pen trails*. Anschließend schneiden wir dieses Kostüm aus dem gelben Block heraus.



Zuletzt wollen wir ein Ellipsoid auf den gelben Block zeichnen. Wir verpassen dem zweiten Sprite ein entsprechendes Kostüm und „pasten“ es auf den gelben Block.





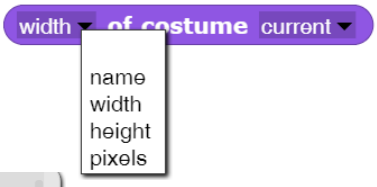


9.4 Zeichnen auf Kostümen – mit einer eigenen JavaScript-Bibliothek

Altersstufe: *Sekundarstufe II* Material: *Color cube on costume*

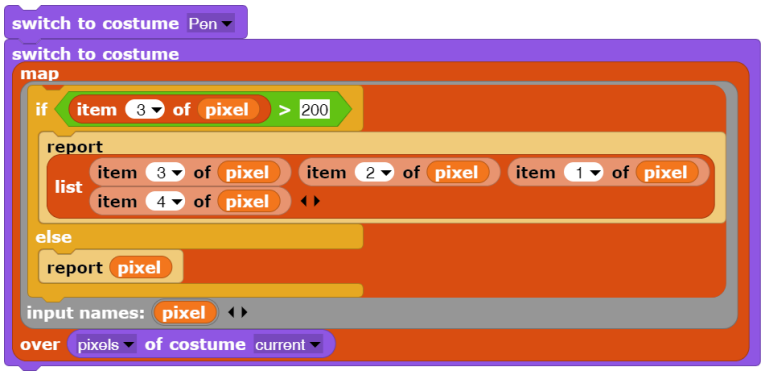
Snap! legt ursprünglich ähnlich wie *Scratch* das *HSV-Farbmodell*[52] zugrunde.[53] Ich bevorzuge aber das *RGB-Modell*[54], weil es direkt den Farbsensoren im menschlichen Auge und vielen technischen Anwendungen entspricht. Vielleicht aber auch aus Gewohnheit. 😉 Inzwischen unterstützt *Snap!* in den Blöcken sowohl *HSV*- wie *RGB*-Darstellungen von Farben.

Mithilfe des *<property> of costume <costume>*-Blocks erhalten wir Zugriff auf die Eigenschaften eines Kostüms, z. B. auf seine Maße und seine Pixel. Als Beispiel nehmen wir unseren Stift. Dessen Pixel sehen wir uns etwas genauer an.



7200	A	B	C	D
141	0	0	0	0
142	0	0	0	0
143	0	0	0	0
144	0	0	0	0
145	73	72	70	255
146	110	99	40	255
147	175	152	23	255
148	240	207	22	255

Wir erhalten eine Liste, die als Elemente 4-elementige Listen mit den drei RGB-Werten der Pixel sowie deren Transparenz-(Alpha)-Werten enthält. Alle Werte stammen aus dem Bereich 0…255, sind also jeweils durch ein Byte darstellbar. Für die Transparenz[55] bedeutet der Wert 0, dass das Pixel „unsichtbar“ ist, und 255, dass es mit vollen Farben gezeichnet werden soll. Mithilfe dieser Pixel-Liste wollen wir den Stift jetzt neu färben. Wir vertauschen deshalb die Blauwerte mit den Rotwerten, aber nur, wenn das Pixel „ziemlich blau“ ist.

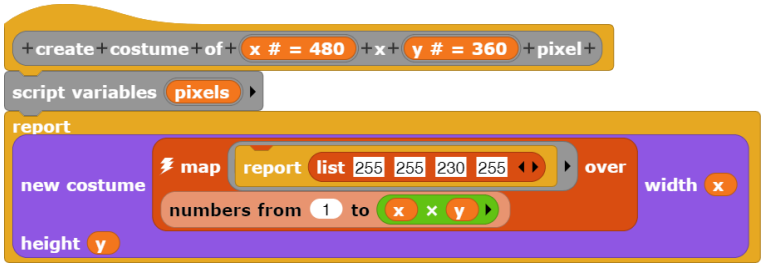


Geht doch!

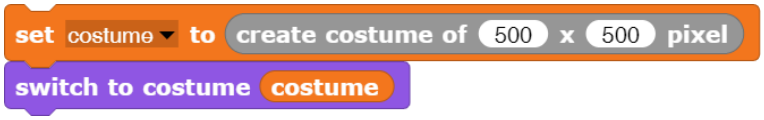
[52] https://de.wikipedia.org/wiki/HSV-Farbraum
[53] In den Bibliotheken von Snap! finden sich weitere Farbmodelle.
[54] https://de.wikipedia.org/wiki/RGB-Farbraum
[55] eigentlich besser: Sichtbarkeit

Das Zeichnen auf Kostümen hat u. a. den Vorteil, dass *JavaScript*-Befehle, die sich auf diesen Bereich beziehen, ohne Kenntnis und Rücksicht auf den restlichen *Snap!*-Bereich eingesetzt werden können. Man hat damit bei Bedarf eine kleine Spielwiese, auf der innerhalb der grafischen Sprache *Snap!* Teile in der textbasierten Sprache *JavaScript* geschrieben werden können.[56] Als Beispiel erzeugen wir wieder den Farbwürfel, diesmal aber auf einem Sprite-Kostüm.

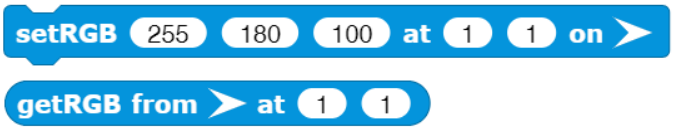
Zuerst einmal benötigen wir ein Kostüm: schwach gelb und ausreichend groß. Wir stellen die Bühne auf die Maße 600x600 Pixel ein und schreiben einen schnellen Block dafür.

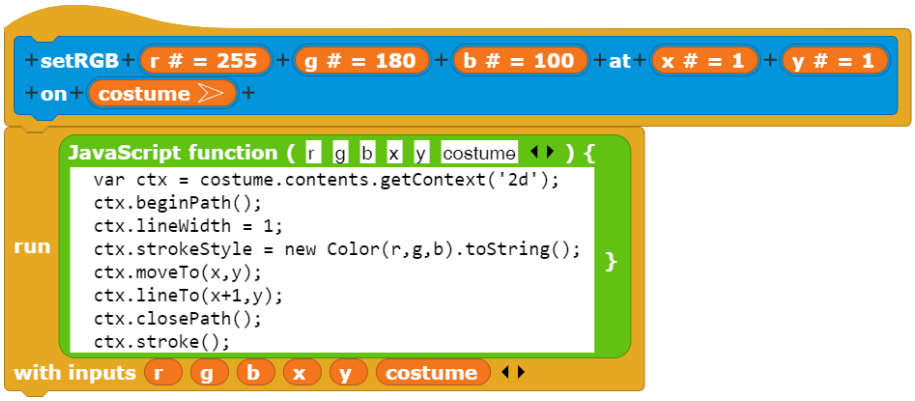


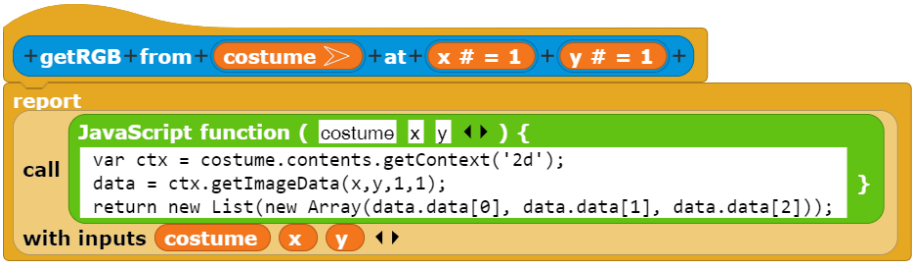
Nach der Erzeugung der entsprechenden Variablen haben wir so den Anfang unseres Skripts gefunden.



Die Pixel dieses Kostüms können wir jetzt manipulieren. Wir schreiben dafür zwei kleine *JavaScript*-Methoden, um die Farbe eines Pixels zu lesen bzw. zu setzen.

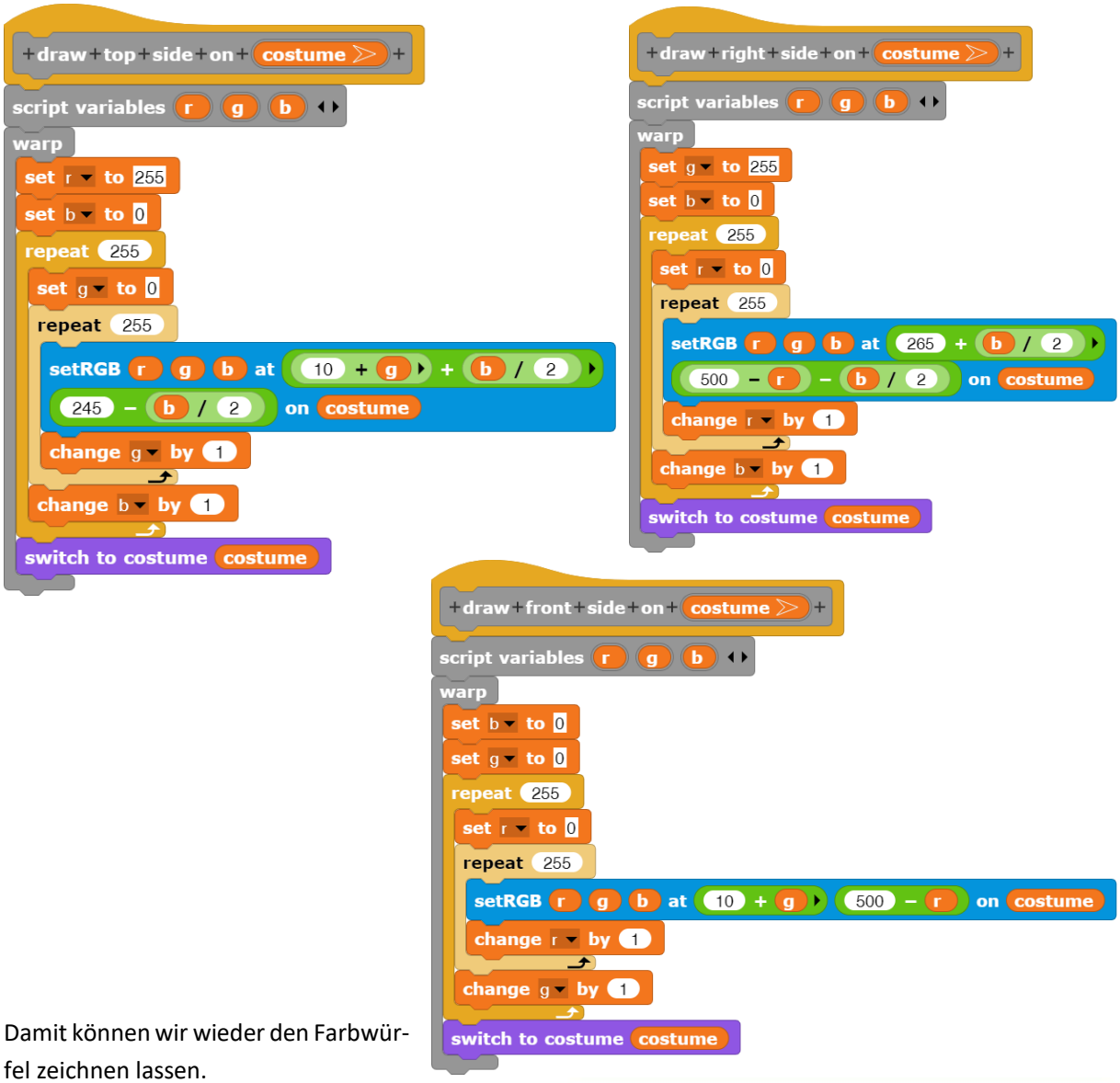




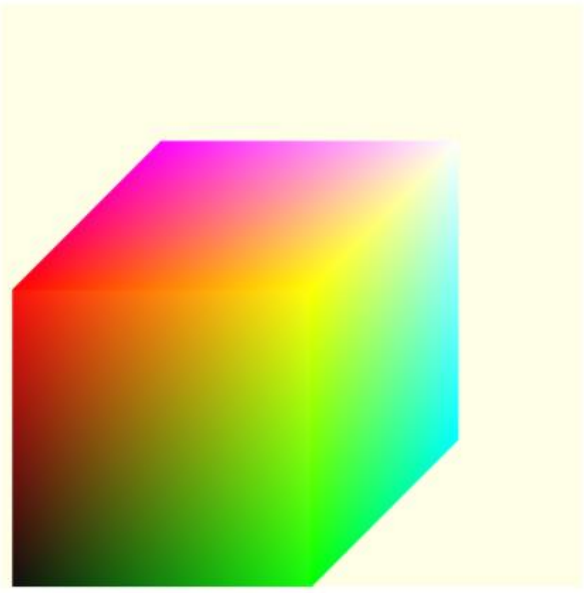


[56] Wenn man das will. Snap! ist inzwischen schnell genug, dass auf solche Erweiterungen verzichtet werden kann.

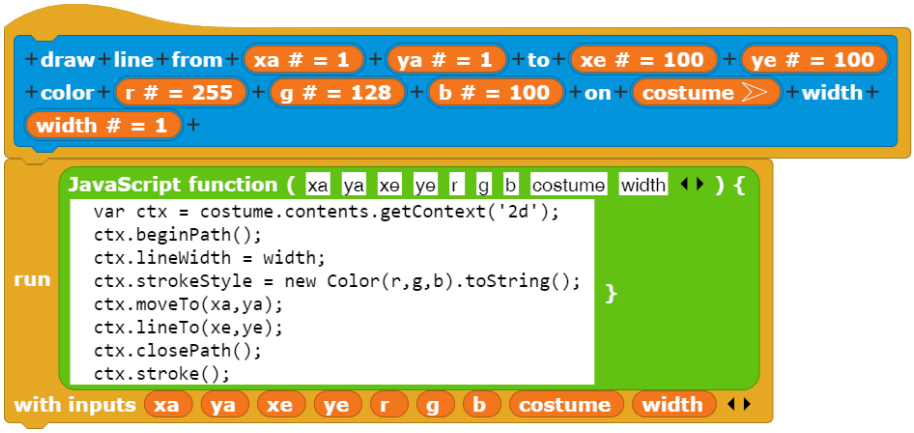
Mit diesen beiden Methoden können wir jetzt wieder den RGB-Farbwürfel erzeugen, nachdem wir im *Settings*-Menü die Benutzung von *JavaScript* zugelassen haben.

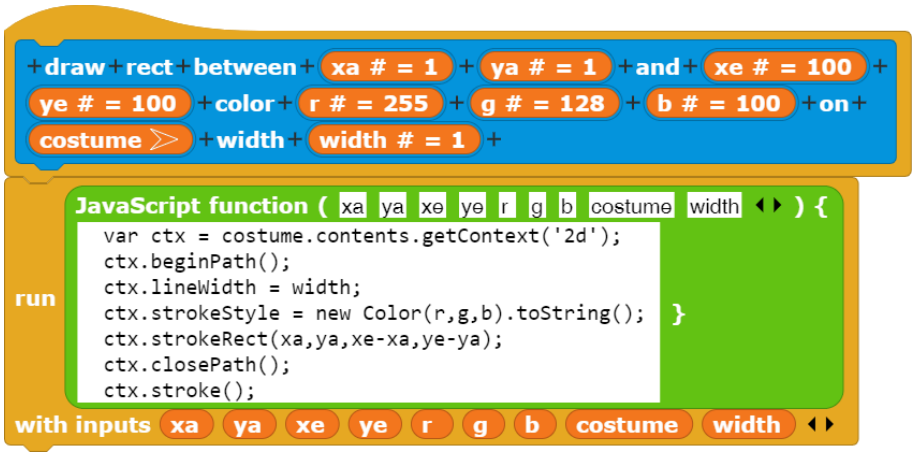


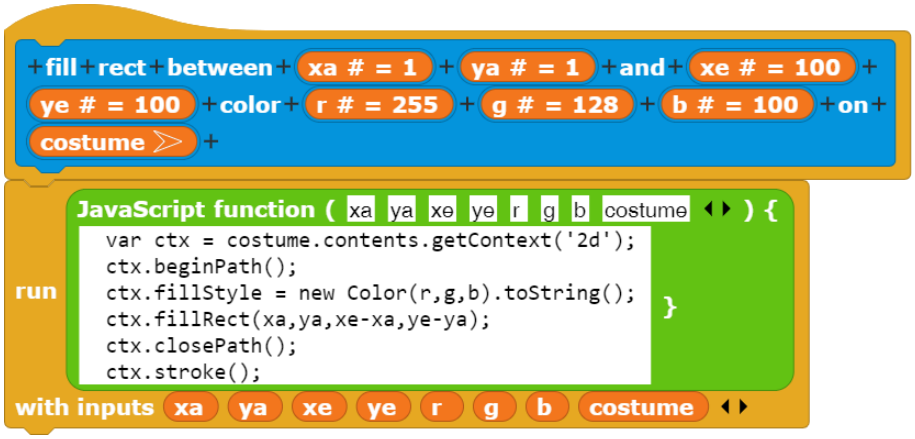
Damit können wir wieder den Farbwürfel zeichnen lassen.



Wenn wir schon einmal dabei sind, können wir auch gleich noch einige weitere der üblichen Grafik-Operationen in *JavaScript* implementieren.



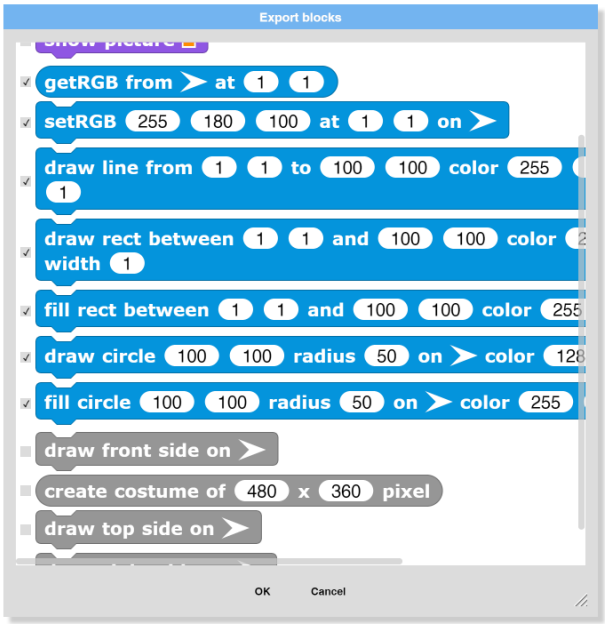




```
+draw+circle+ x # = 100 +y # = 100 +radius+ radius # = 50 +on+
costume >> +color+ r # = 128 + g # = 100 + b # = 100 +width+
width = 1 +
run JavaScript function ( x y radius costume r g b width ) {
    var ctx = costume.contents.getContext('2d');
    ctx.beginPath();
    ctx.lineWidth = width;
    ctx.strokeStyle = new Color(r,g,b).toString();
    ctx.arc(x,y,radius,0,6.283185307179586476925286766559);
    ctx.closePath();
    ctx.stroke();
}
with inputs x y radius costume r g b width
```

```
+fill+circle+ x # = 100 +y # = 100 +radius+ radius # = 50 +on+
costume >> +color+ r # = 255 + g # = 0 + b # = 0 +
run JavaScript function ( x y radius costume r g b ) {
    var ctx = costume.contents.getContext('2d');
    ctx.beginPath();
    ctx.fillStyle = new Color(r,g,b).toString();
    ctx.arc(x,y,radius,0,6.283185307179586476925286766559);
    ctx.fill();
    ctx.closePath();
    ctx.stroke();
}
with inputs x y radius costume r g b
```

Diese Blöcke speichern wir in einer eigenen Bibliothek (*File → Export blocks…*), wobei wir vorher auswählen, welche Blöcke darin aufgenommen werden sollen. Wir benennen die im Download-Verzeichnis gespeichert Datei um, z. B. in *MyOwnDrawingLibrary.xml* und verschieben Sie an eine geeignete Stelle. Von dort können wir die Blöcke über *File → Import…* in andere Projekte laden und benutzen – wie jede andere Bibliothek auch.

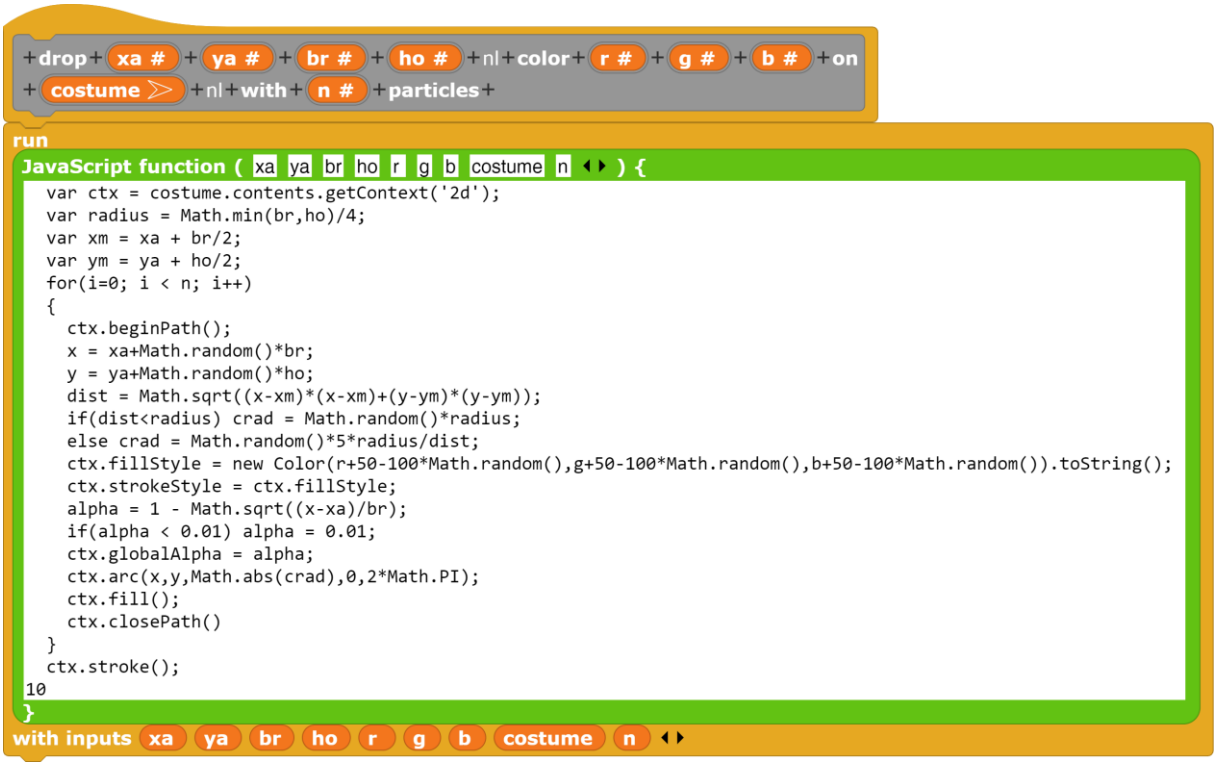


9.5 Drip Painting

Altersstufe: *Sekundarstufe II* Material: *Drip painting*

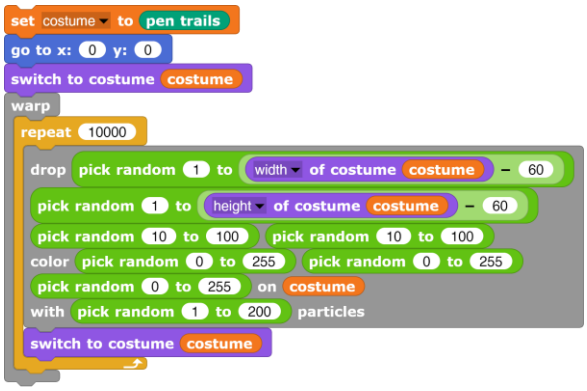
Eines der Verfahren, in der modernen Malerei den Zufall in die künstlerische Gestaltung zu bringen, besteht darin, mit dem Pinsel Farbkleckse auf der Leinwand zu verspritzen. Die auftreffenden Farbtropfen werden beim Aufprall weiter geteilt, sodass sich ein Zufallsmuster ergibt. Wir wollen dieses Verfahren, das *Drip painting*, simulieren – und das ist gar nicht so leicht.

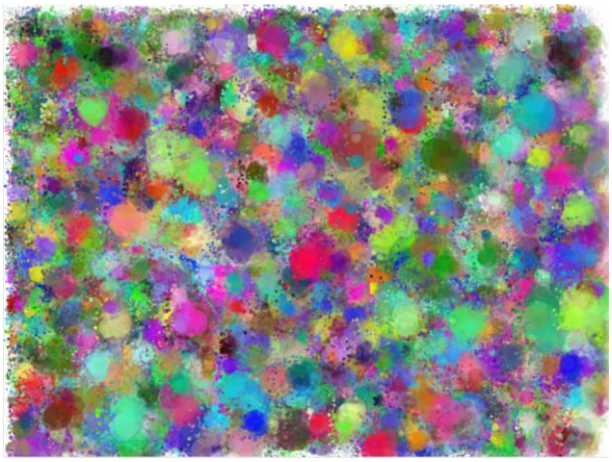
Wir versuchen es mit einem einfachen, aber sehr rechenaufwändigen Ansatz: Innerhalb eines Rechtecks werden *n* zufällige Kreisflecken mit leicht unterschiedlichen Farben erzeugt, die zu den Rändern des Rechtecks hin transparenter werden. Schließlich nimmt dort die Farbdicke ab. Da *n* in der Größenordnung von 100 liegt und wir einige tausend Tropfen pro Bild verteilen wollen, übertragen wir das Tropfenzeichnen einer JavaScript-Funktion, die so etwas sehr schnell erledigen kann.



Als Parameter übergeben wir die Koordinaten der oberen-linken Ecke im ebenfalls übergebenen Kostüm, die Breite und Höhe des den Tropfen umschreibenden Rechtecks, die drei RGB-Farbwerte und die Anzahl der „Teiltropfen“. In der Funktion wird (wie inzwischen bekannt) der 2D-Grafikkontext bestimmt und ein Radius für den Kernbereich des Tropfens berechnet. Danach werden die Koordinaten der Bildmitte ermittelt und *n* Teiltropfen gezeichnet, deren Positionen, Radien, Farben und Transparenz zufallsgesteuert gewählt werden. Ein stark vergrößerter „Tropfen“ sieht dann z. B. so aus:

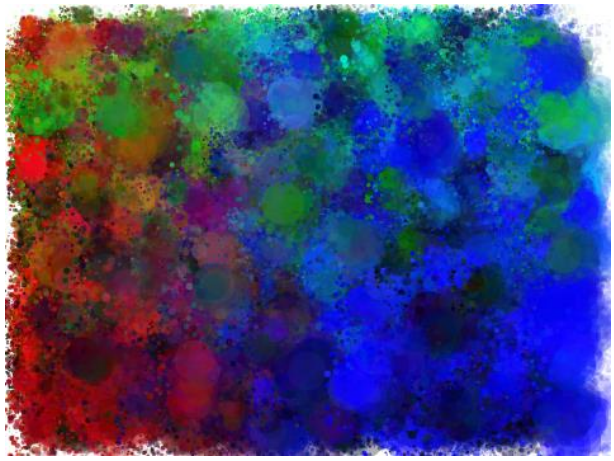
Von diesen Tropfen verteilen wir jetzt einige tausend auf der Leinwand – und erhalten ein optimistisches abstraktes *Frühlingsbild*.



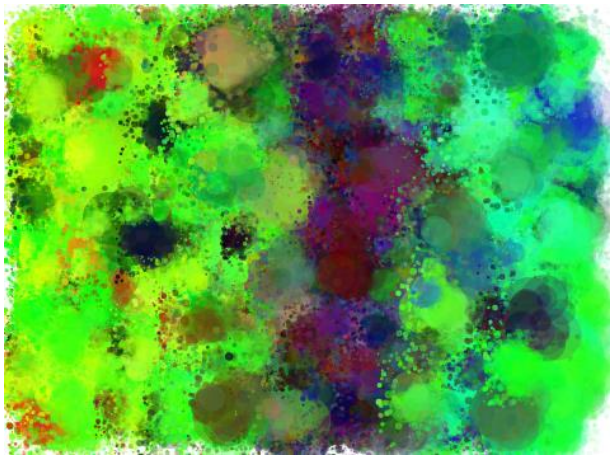


Aber wir können natürlich auch die Farbverteilung von der Position abhängig machen – und erhalten *Rot und viel Blau*.

Mit etwas Grün dazu: *Ohne Titel 37*.



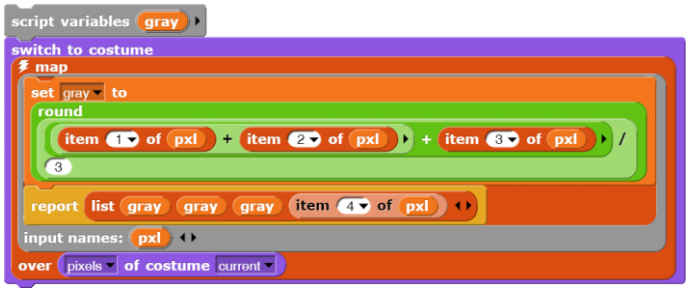
Und man kann natürlich auch mutiger werden: *Gratwanderung* 😉



9.6 Kantenerkennung

Altersstufe: *Sekundarstufe II* Material: *Edge detection*

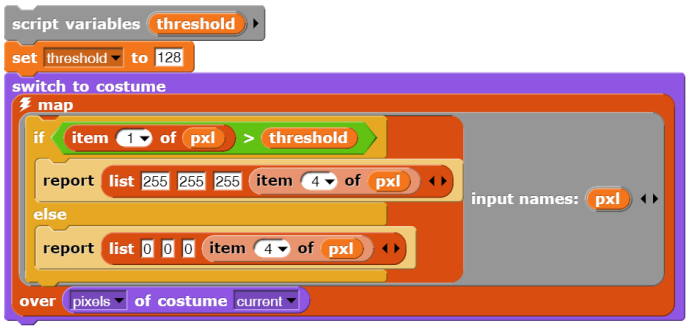
Um Gegenstände auf einem Bild zu erkennen, ist es oft hilfreich, die Begrenzungen dieser Gegenstände hervorzuheben – eben die *Kanten*. Ein mögliches Verfahren dafür besteht aus den Schritten 1) Umwandlung in ein *Graustufenbild*, 2) mithilfe eines Schwellwertes Umwandlung in ein *Schwarzweißbild* und 3) *Kantenerkennung* in diesem Schwarzweißbild. Die ersten beiden Schritte lassen sich in *Snap!* relativ schnell mithilfe der *map…over*–Funktion ausführen, beim dritten ist der Rechenbedarf sehr hoch, sodass sich reichlich Gelegenheiten für Kaffeepausen ergeben. Oder wir übertragen diese Aufgabe, nachdem wir das Verfahren in *Snap!* entwickelt haben, an eine *JavaScript*-Funktion. Kantenerkennung ist eine Vorstufe zur Gegenstandserkennung. Als Beispiel kann die Erkennung des Nummernschilds eines Kraftfahrzeugs auf einem Videobild dienen.



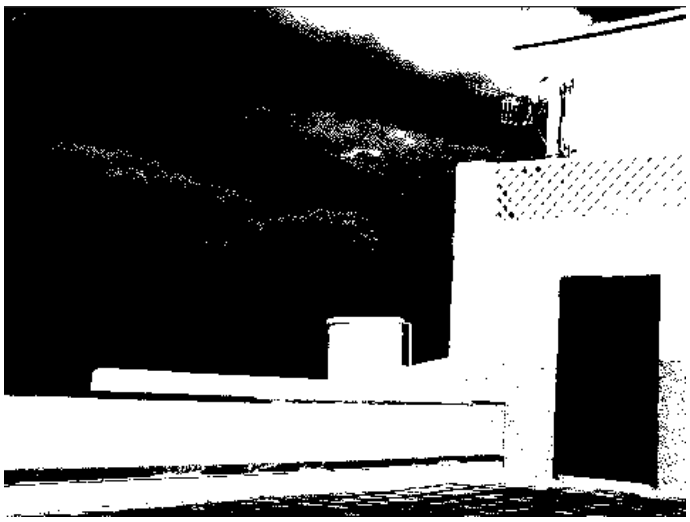
Wir besorgen uns ein Bild, auf dem sich gut sichtbare Kanten befinden, und laden es als Kostüm unseres Sprites. Danach wechseln wir zu einer Kopie des Kostüms, um das Original zu schonen. Breite, Höhe und die Pixelliste des Bildes ermitteln wir mit dem dafür vorgesehenen Block aus der *Looks*-Palette.

Dieses Bild soll in ein Graustufenbild umgewandelt werden. Das können wir Schritt für Schritt erreichen, indem wir die einzelnen Pixel bearbeiten – eine typische Aufgabe für die *map…over*-Funktion, hier in der vorkompilierten Version. Diese benötigt ein Skript, das sie auf die einzelnen Listenelemente anwenden kann. Es berechnet den Mittelwert *gray* der drei RGB-Werte und weist diesen den drei Farbkanälen zu. Den Transparenzwert lässt es unverändert.

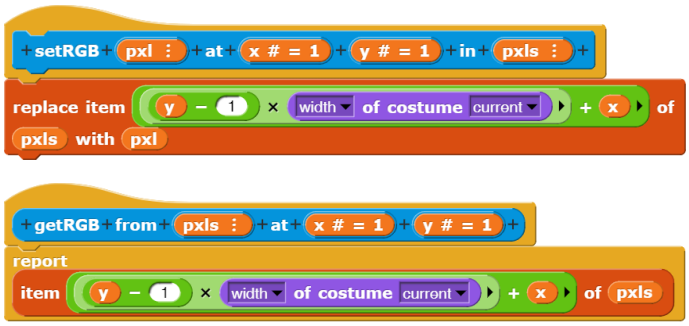
Aus dem Graustufenbild soll ein Schwarzweißbild erstellt werden. Dazu geben wir einen Schwellwert (*threshold*) an. Alle Grauwerte, die größer als der Schwellwert sind, werden auf volles Weiß gesetzt, die anderen auf Schwarz. Auch hierfür schreiben wir eine Funktion, die von *map…over* ausgeführt wird.



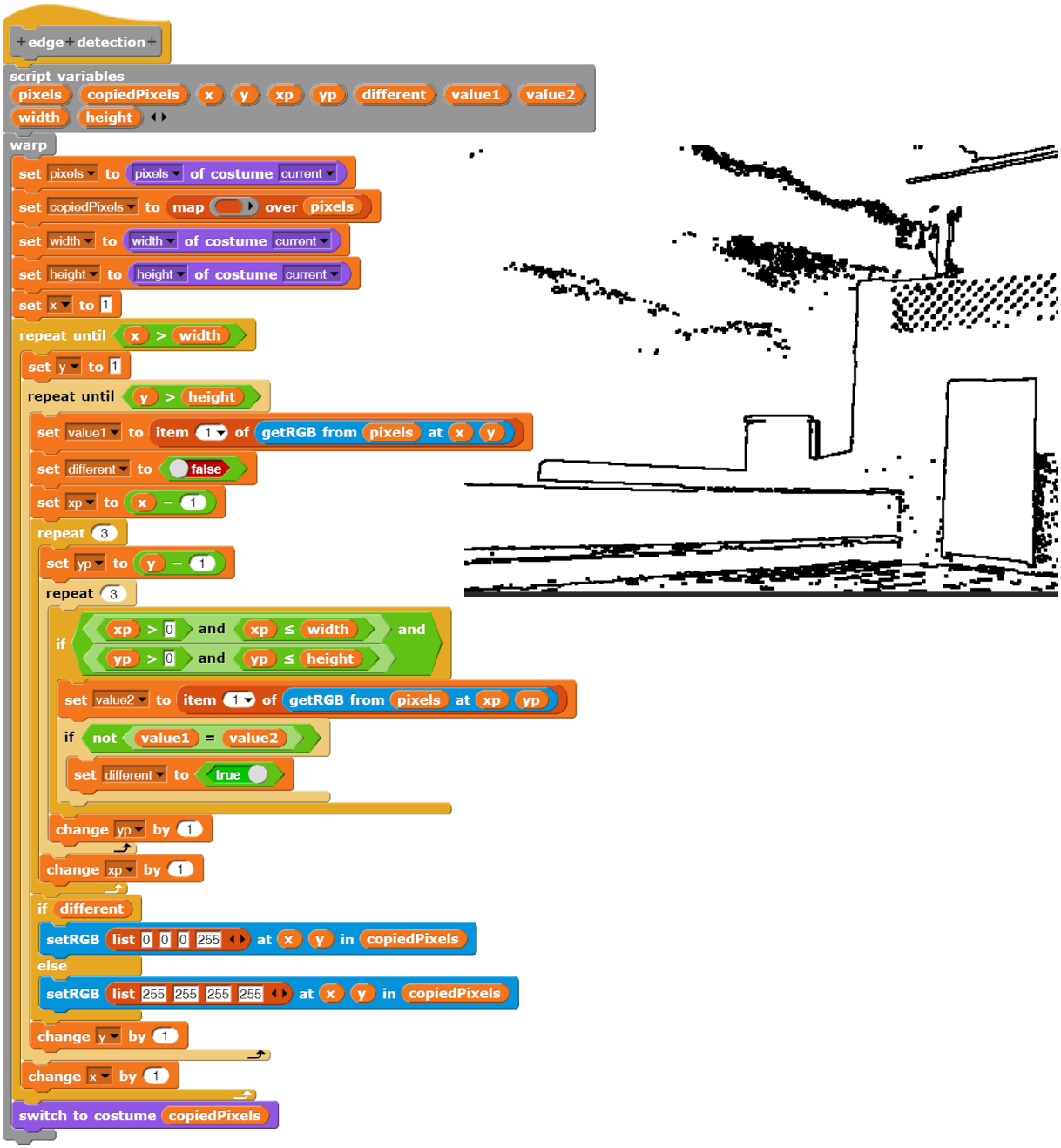
In dem Schwarzweißbild sollten noch einige Reparaturarbeiten erfolgen: einzelne isolierte Punkte sollten gelöscht, Linienlücken geschlossen werden usw. (s. Aufgaben). Darauf verzichten wir hier.



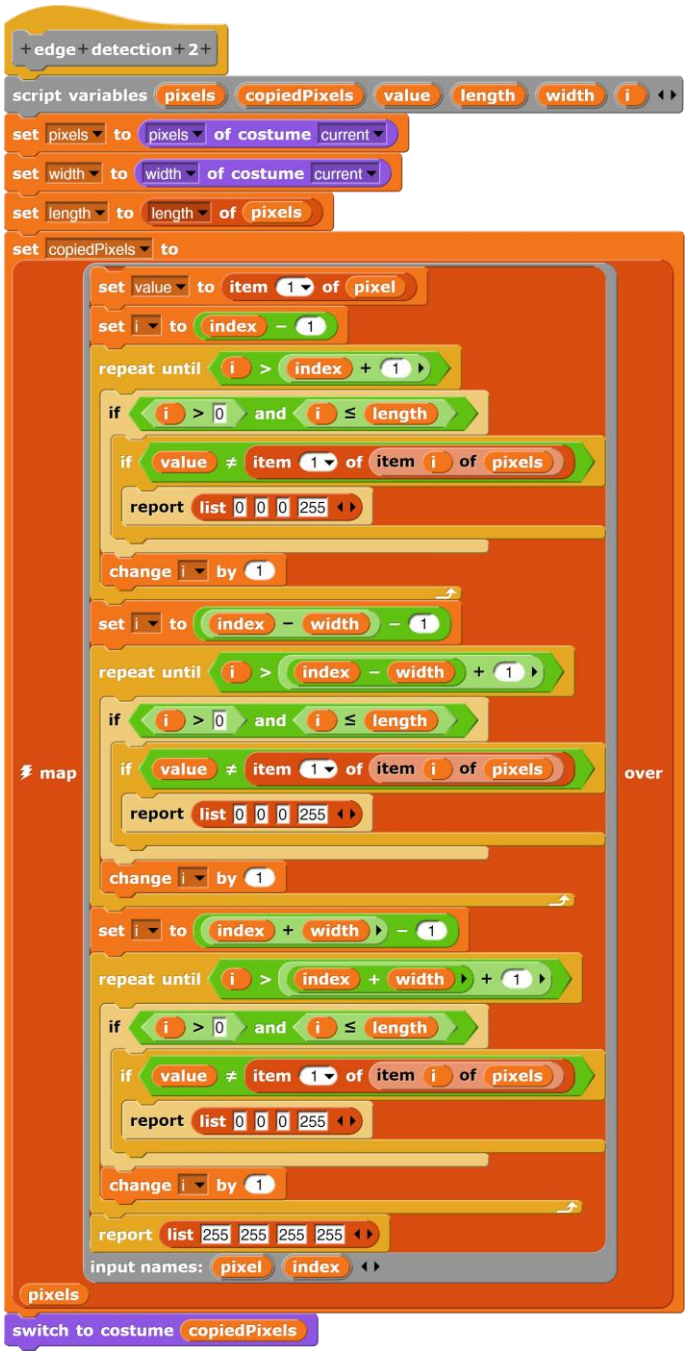
Im letzten Schritt werden die Kanten im Schwarzweißbild gesucht. Dazu untersuchen wir das Umfeld jedes Pixels. Haben alle Punkte die gleiche Farbe wie das Pixel, dann befindet sich dieses in einer Fläche und wird weiß gezeichnet. Findet sich wenigstens ein anders gefärbtes Pixel, dann haben wir ein Randpixel gefunden und färben es schwarz. Da sich Änderungen der Pixelwerte auf die Nachbarschaft auswirken, erfolgen die Änderungen in einer Liste *copiedPixels*.



Zuerst einmal müssen wir Zugriff auf die einzelnen Pixel über deren Koordinaten im Bild haben. Wir könnten dafür die vorher entwickelte *JavaScript*-Grafikbibliothek benutzen, wollen aber hier zwei kleine Blöcke dafür erstellen. Die benutzen wir dann in einem Block *edge detection* zur Kantenerkennung.



In *edge detection* haben wir sehr traditionell die Umgebungen aller Punkte untersucht, was entsprechend lange dauert. Wir können aber genauso gut die Pixelliste sequentiell untersuchen und dabei berücksichtigen, dass wir die Nachbarn eines Pixels teilweise neben dem Pixel selbst finden, teilweise um eine Bildbreite „nach links“ bzw. „nach rechts“ verschoben. Der sequentielle Durchlauf ermöglicht die Benutzung des *map…over*-Blocks in der vorkompilierten Version. Wir wollen mal sehen, ob sich der Aufwand dafür lohnt. Der Kürze halber verzichten wir hier auf die Prüfung, ob wir ein Randpixel vorliegen haben. Wir behandeln also das Bild als Torus.



Die drei Pixel in der Mitte untersuchen.

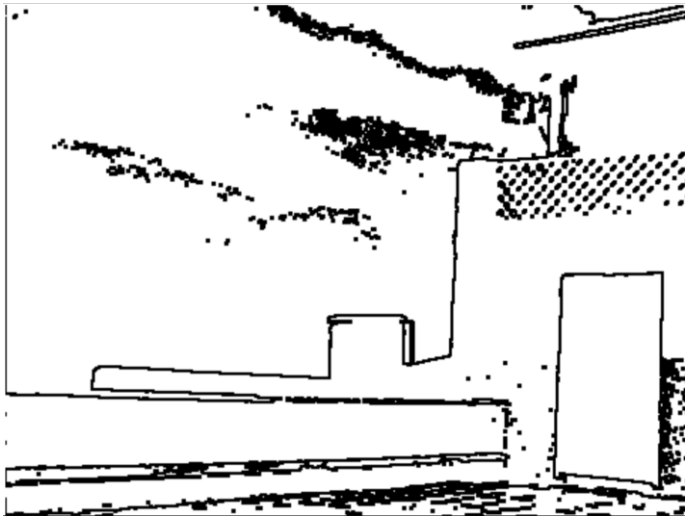
Wenn ein Farbunterschied gefunden wurde, ein schwarzes Pixel zurückgeben.

Jetzt die in der oberen Reihe über dem betrachteten Pixel, …

… und dann die drei darunter.

Wenn keine Abweichungen gefunden wurden, ein weißes Pixel zurückgeben.

Wir erhalten - mit kleinen Abweichungen an den Rändern - dasselbe Bild wie vorher. Allerdings dauerte die Bearbeitung diesmal nur halb so lange.



9.7 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 *Snap!*.
   d: Verfahren Sie entsprechend für die **Dragon**-Kurve, die **Peano**-Kurve und die **Sierpinski**-Kurve.

2. Stellen Sie den RGB-Würfel aus **anderer Sicht** so dar, dass die drei bisher verdeckten Seiten sichtbar werden.

3. Falls Sie sich etwas JavaScript versuchen wollen: erzeugen Sie Farbverläufe und ggf. auch den RGB-Farbwürfel in einer **JavaScript-Funktion**.

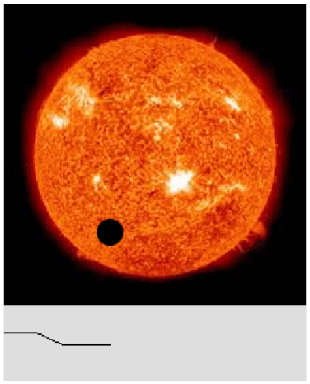
4. Ändern Sie die Farbwerte iterativ, also ohne die *Map*-Funktion, indem Sie auf die einzelnen Pixel zugreife. Messen Sie die **Ausführungszeiten** bei den unterschiedlichen Verfahren.

5. Manche Maler tragen die Farben mit einem Spachtel auf. Erzeugen Sie „**Spachtelbilder**“, die in eine Richtung „auslaufen“ und mehrere Farben enthalten können. Erzeugen Sie daraus Zufallsbilder.

6. a: Löschen Sie in Schwarzweißbildern einzelne **isolierte Pixel**.
   b: Wenn man in Schwarzweißbildern alle **Randpunkte** löscht (die Kanten „abschmilzt“) und danach wieder an alle Randpunkte Punkte anfügt – oder umgekehrt, dann kann man durch abwechselndes und ggf. wiederholtes Anwenden der Verfahren Einzelpixel löschen, Lücken in Linien schließen usw. Implementieren Sie die Verfahren und probieren Sie sie aus.

7. Falls Sie etwas in JavaScript programmieren wollen:
   a: Implementieren Sie die Umwandlung von **Graustufenbildern** in Schwarzweißbilder als JavaScript-Funktion. Der Schwellwert soll durch eine Variable in Slider-Darstellung gegeben werden.
   b: Implementieren Sie die **Kantenerkennung** als JavaScript-Funktion.

8. Extrasolare Planeten werden meist entdeckt, wenn sie bei ihrem Durchgang zwischen ihrem Stern und der Erde ihre Sonne etwas verdunkeln. Besorgen Sie sich ein Bild der Sonne und lassen Sie einen schwarzen Kreis, den Planeten, vor der Sonne vorbeiziehen. Zählen Sie jeweils die Anzahl der sichtbaren hellen Pixel und stellen Sie die Ergebnisse des **Planetentransits** in einem Diagramm dar.



10 Bilderkennung

Die folgenden drei Beispiele stellen eine Folge dar, in der mit zunehmendem Schwierigkeitsgrad einige Möglichkeiten von *Snap!* in der Bildbearbeitung gezeigt werden. Es wurden Probleme gewählt, die Zugang zur aktuellen Diskussion der digitalen Medien liefern und die somit für den Bereich *Informatik und Gesellschaft* (IuG) relevant sind.

10.1 Ein Barcodescanner[57]

Altersstufe: *Sekundarstufe I* Material: *Barcode reader*

Wir wollen einen Barcode (Strichcode), wie er auf den Etiketten der Waren in einem Supermarkt benutzt wird, mithilfe eines „Lasers“ (eines roten Punktes) analysieren und in eine Zeichenkette umsetzen. Zuerst einmal sehen wir uns den geplanten Aufbau an, dabei sollten wir den sehr kleinen roten Punkt links im Arbeitsbereich nicht übersehen – das ist der „Laser“!

Was ist ein EAN-Code?

Den Code der Europäischen Artikel-Nummern (EAN) gibt es in verschiedenen Varianten. Wir betrachten hier den EAN-8-Code, der aus 8 Ziffern besteht, wobei die letzte eine Prüfziffer darstellt[58]. Die Ziffern werden durch jeweils vier unterschiedlich breite schwarze und weiße Streifen dargestellt. Der Zwischenraum zwischen zwei schwarzen Strichen gehört also auch zum Code! Links und rechts vom Barcode befinden sich jeweils zwei schwarze und ein weißer Streifen dazwischen als Begrenzer. Die Mitte wird durch fünf solcher Striche markiert. Alle haben die Breite „1“. Der Code wurde so gewählt, dass alle Ziffern insgesamt die Breite „7“ haben. Auf weitere Feinheiten gehen wir hier nicht ein.

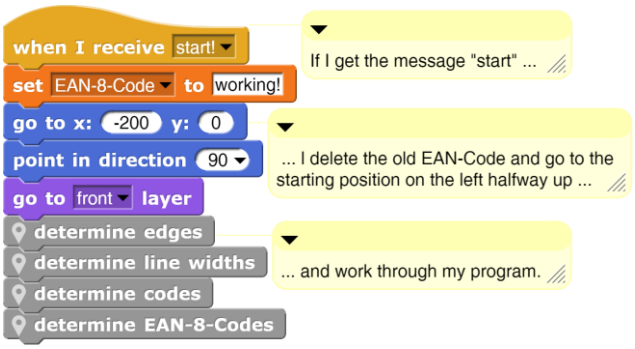
EAN-8-Codetabelle	
Ziffer	Code
0	3211
1	2221
2	2122
3	1411
4	1132
5	1231
6	1114
7	1312
8	1213
9	3112

Zur Ermittlung der kodierten Zahlen wird der Laserpunkt von links nach rechts über den Code geführt. Er „misst“ dabei die Positionen der Farbwechsel und trägt sie in eine Liste ein. Aus dieser werden die Strichbreiten berechnet. Da die ersten drei Striche die Breite „*1*“ haben, können wir durch Mittelwertbildung diesen Wert ganz gut bestimmen. Die anderen Strichbreiten ergeben sich dann als Vielfache dieser Einheit. Jeweils vier Striche ergeben den Code einer Ziffer, die wir anhand der Tabelle bestimmen. Das Verfahren lässt sich knapp in Form eines Struktogramms zusammenfassen:

Bestimme die x-Positionen der Ränder der schwarzen und weißen Linien
Bestimme aus diesen die Strichbreiten, lösche dabei die Markierungen
Bestimme aus diesen die acht vierstelligen Codes der Ziffern
Bestimme aus diesen den EAN-Code

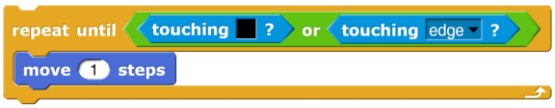
[57] Teilweise nach E. Modrow, The SQLsnap supermarket, Scratch2015 Amsterdam
[58] Siehe z. B. https://de.wikipedia.org/wiki/European_Article_Number

Umgesetzt in ein *Snap!*-Skript des Lasers erhalten wir:

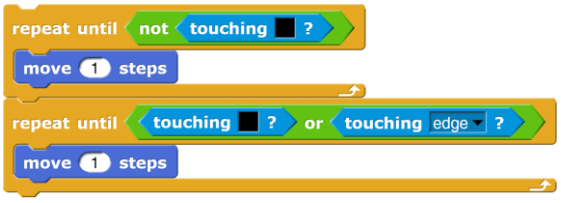


Dafür haben wir in der *Variables*-Palette von *Snap!* den Button *„Make a variable“* gedrückt, im aufklappenden Fenster den Variablennamen *EAN-8-Code* eingegeben und diese Variable als lokal (*„for this sprite only“*) markiert. Da sie bei keinem anderen Objekt benötigt wird, beschränken wir ihre Gültigkeit auf die Skripte des Lasers. Danach erscheint die Variable in der *Variables*-Palette. Weil wir schon mal dabei sind, erzeugen wird auch gleich drei andere Variable mit den Namen *edges*, *line widths* und *encoding*. Das Häkchen vor der *EAN-8-Code*-Variablen bedeutet, dass die Variable im Ausgabefenster angezeigt wird. Dort können wir ihr Aussehen im Kontextmenü (Rechtsklick auf die Variable) noch verändern. Den ersten Block unter den Variablennamen *set <variable> to <wert>* ziehen wir in den Skriptbereich. Mithilfe des kleinen schwarzen Pfeils können wir dann einen für den Laser „sichtbaren“ Variablenbezeichner auswählen und für diesen einen Wert angeben. Klicken wir dann den Block an, so wird er ausgeführt und die Variable erhält den gewünschten Wert, was man im Ausgabebereich sofort sieht.

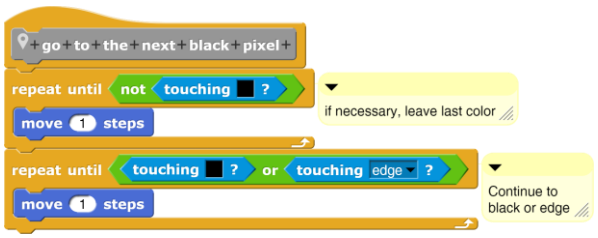
Nach diesen Vorbereitungen müssen wir langsam mal anfangen, das eigentliche Problem zu lösen. Eines müssen wir dem Laser in jedem Fall beibringen: den nächsten schwarzen Strich zu finden. Wir wechseln dazu in den *Costumes*-Bereich und zeichnen dort einen kleinen roten Punkt als neues Kostüm – den Laserpunkt. Alternativ dazu können wir das Kostüm auch mit einem Grafikprogramm erstellen, als *png*-Datei speichern und in den *Costumes*-Bereich ziehen. Mithilfe des Blocks *touching <color>* aus der *Sensing*-Palette können wir jetzt prüfen, ob unser Laser-Sprite die angegebene Farbe berührt. Diese Farbe können wir nach Anklicken des Farbfeldes irgendwo aus dem *Snap!*-Fenster oder aus dem aufklappenden Farbfeld wählen. Diesen Block sowie einen zweiten, der feststellt, ob der Rand des Arbeitsbereichs erreicht wurde, nutzen wir als Abbruchbedingung einer Schleife aus der *Control*-Palette, in der das Laser-Sprite jeweils einen Schritt nach rechts bewegt wird.

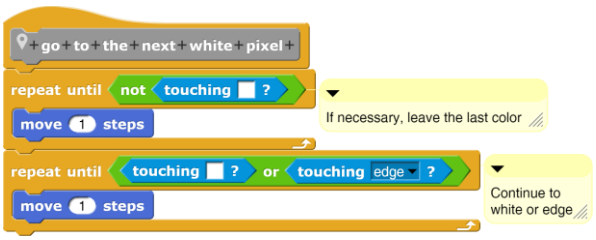


Beim Testen dieses Blocks stellen wir fest, dass sich der Laser manchmal gar nicht bewegt. Beim wiederholten Überlaufen der Striche wird es passieren, dass der Laser einerseits einen weißen Streifen berührt, andererseits aber immer noch auch einen schwarzen. Schließlich hat er ja eine, wenn auch geringe, Ausdehnung. Wir sorgen deshalb dafür, dass er zuerst einmal soweit vorrückt, dass er keine schwarzen Bereiche mehr tangiert. Dann läuft er los.



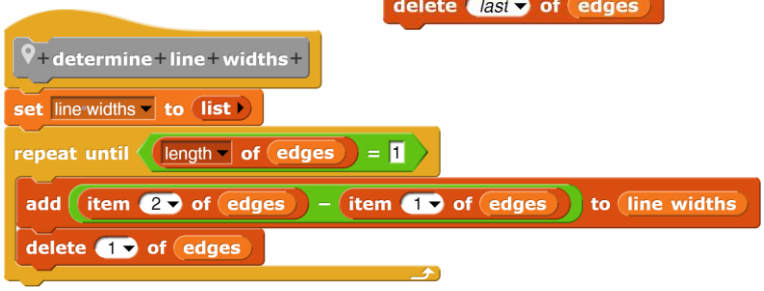
Nach dem ausführlichen Testen dieses Skripts verpacken wir es in einer eigenen Methode, einem neuen Block, namens *go to the next black pixel*, die als lokal gekennzeichnet wird, weil sie niemand sonst benötigt. Danach erzeugen wir eine ganz ähnliche Methode *go to the next white pixel*. Die Kommentarblöcke findet man im Kontextmenü nach Rechtsklick auf den Skriptbereich.

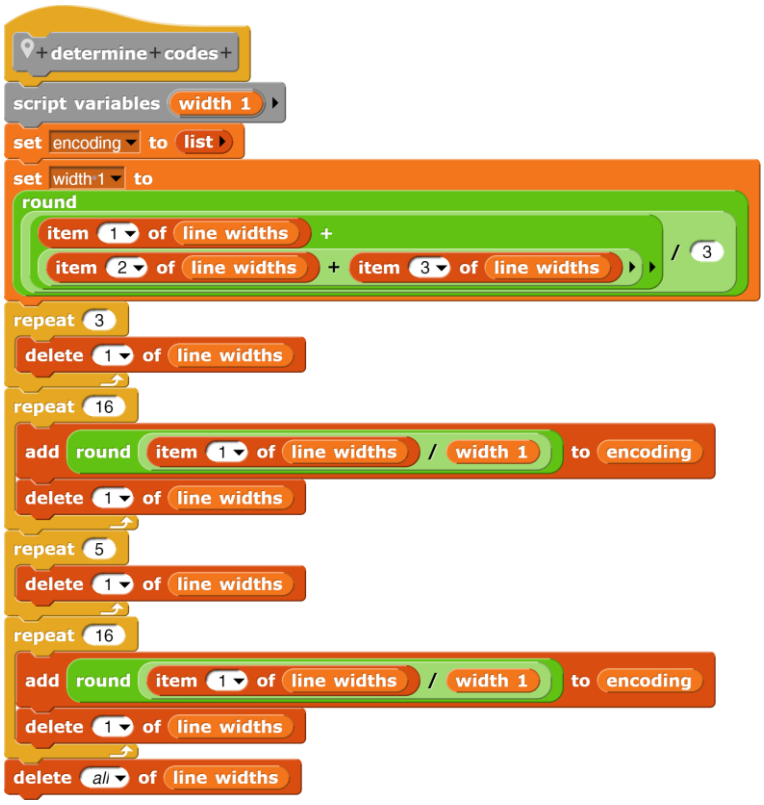




Das Zusammenspiel dieser beiden Methoden testen wir ausführlich. Danach sorgen wir dafür, dass die Variable *edges* als Wert eine leere Liste erhält (*set <edges> to <list>*) und dass jeweils nach Erreichen eines neuen Strichs die x-Position des Lasers dieser Liste hinzugefügt wird (*add <x-position> to <edges>*). Die letzten beiden Werte dieser Liste löschen wir, da sie beim Erreichen des Randes erzeugt werden. Die Arbeit dieses Skripts können wir beobachten, wenn wir *edges* mit einem kleinen Häkchen als sichtbar markieren. Da alles klappt, wird das Skript in einem neuen Block *determine edges* verpackt.

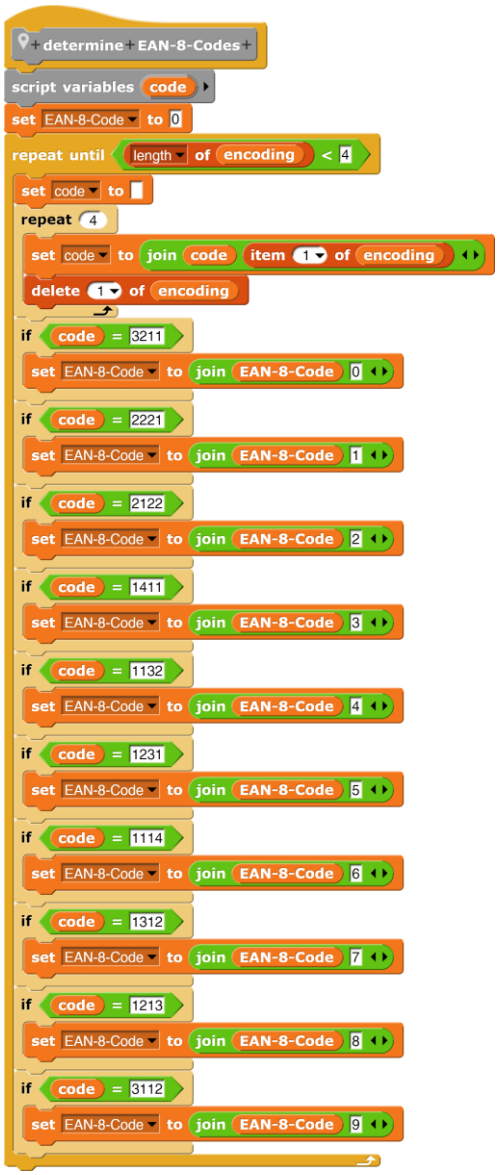
Jetzt folgen drei sehr ähnliche Methoden, in denen jeweils die letzte gerade erzeugte Liste durchlaufen wird, um die nächsten Werte zu bestimmen. Wie verarbeiten jeweils die ersten Werte der Listen und löschen sie dann, bis wir „durch sind“.



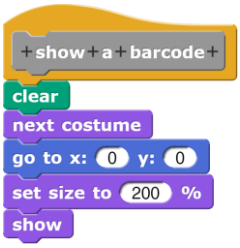


Zuerst einmal berechnen wir die Breiten der abgetasteten Striche als Differenzen der Werte der *edges*-Liste und speichern sie in der *line widths*-Liste. Danach bestimmen wir die dadurch dargestellten Codes, indem wir die Breite „*1*“ aus den ersten drei Strichbreiten mitteln und sie in der Skriptvariablen *width 1*, die nur innerhalb des neuen Blocks bekannt ist, speichern. Wir löschen dann die Anfangsmarkierung und berechnen die ersten 16 Strichbreiten für die ersten vier Zahlen. Dann löschen wir die Mittelmarkierung und gehen entsprechend für die zweiten vier Zahlen vor. Dann wird der Rest der Liste *strichbreiten* gelöscht. Die ermittelten Werte werden in der Liste *encoding* gespeichert.

Jetzt fehlt nur noch die Dekodierung der Zahlenwerte in der *encoding*-Liste. Wir vereinbaren wieder eine Skriptvariable *code* für den neuen Block. Diese setzen wir wiederholt aus vier Zahlenwerten zusammen (mit dem *join*-Block aus der *Operators*-Palette, der mit Zeichenketten arbeitet). Je nach Wert des Ergebnisses erhalten wir die nächste Ziffer des EAN-Codes.



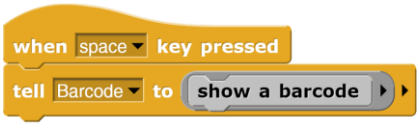
Unsere neuen Blöcke, die wir wie jeden anderen Befehlsblock auf der Laser-Skriptebene benutzen können, finden wir ganz unten in der *Variables*-Palette. Die kleine Markierungsnadel vor den Methodennamen zeigt an, dass die Methoden lokal für das Sprite sind. In anderen Sprites sind sie nicht zu sehen.



Die Barcodes erzeugen wir mit einem der Generatoren dafür im Internet und speichern sie als Kostüme eines neuen Sprites, das wir mit dem Pfeil-Button über dem Sprite-Bereich unten-rechts im Fenster erzeugen. Wir taufen dieses Sprite *Barcode*. Um zwischen den Kostümen umzuschalten, erstellen wir einen globalen Block *show a barcode* (um auch diesen Weg der Kommunikation zwischen Objekten zu zeigen). Dieser vergrößert das Kostüm auf das Doppelte und rückt das Sprite in die Mitte. Der Block ist bei allen Sprites zu sehen.

Unser kleines Projekt soll durch Skripte der Bühne (*Stage*) gesteuert werden. Wenn die grüne Fahne angeklickt wird, dann wird zuerst das *Barcode*-Objekt gebeten, einen neuen Barcode anzuzeigen – also das Kostüm zu wechseln. Das geschieht mit *tell <Barcode> to <show a barcode>*.

Da der auszuführende Block, grau umringt und so als Code gekennzeichnet, global vereinbart wurde, können wir ihn einfach in den vorher leeren Slot im *tell*-Block ziehen. Anschließend schickt die Bühne die Nachricht „start!“ nur an das Laser-Objekt. Alternativ hätte es diese Nachricht auch an alle schicken können. Wenn nur das *Laser*-Sprite reagiert, dann hätte dieses die gleiche Wirkung.



Die beiden letzten Skripte dienen dazu, Kostümwechsel auch durch Drücken der Leertaste und Lesevorgänge durch Klicken auf die Bühne zu veranlassen.

10.2 Projekt: Durchfahrt verboten!

Altersstufe: *Sekundarstufe I/II* Material: *Transit prohibited*

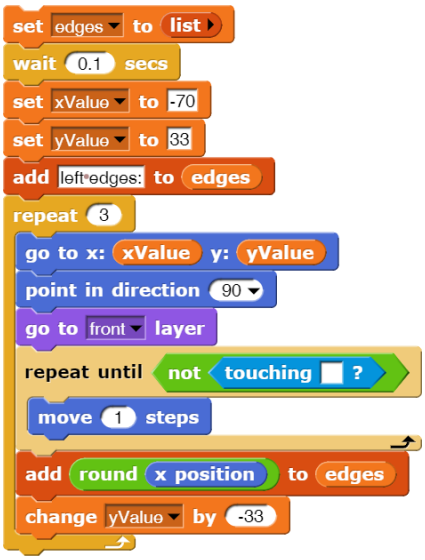
Moderne Autos verfügen über eine Kamera, mit deren Hilfe sie Verkehrsschilder „sehen“ und erkennen können. So etwas wollen wir auch versuchen. Wir suchen uns dazu die Bilder einiger gängiger Verkehrszeichen und skalieren sie mithilfe eines Grafikprogramms alle auf die Größe 100 x 100 Pixel. Dann ziehen wir sie in den *Costumes*-Bereich eines *Snap!*-Sprites, das wir *Traffic sign* nennen.

Wie man sieht, sind die Schilder recht unterschiedlich. Eine Aufgabe wird es deshalb sein, die Form des Schildes zu erkennen. Wir finden *runde*, *rechteckige* und *unterschiedliche dreieckige* Schilder. Zum Glück verfügen wir aus dem letzten Projekt schon über einen *Laser*, den wir für die neue Aufgabe umbauen werden. Wir exportieren dazu das Laser-Sprite aus dem *Barcode*-Projekt in eine XML-Datei *Laser.xml* (Rechtsklick auf das Sprite, „export…“ aus dem Kontextmenü anklicken) und importieren diese Datei in das neue Projekt entweder mithilfe des Datei-Menüs oder indem wir es auf das *Snap!*-Fenster ziehen. In der *Variables*-Palette des Lasers löschen wir alle Variablen bis auf *edges*, dann löschen wir die lokalen Methoden bis auf *go to the next black pixel*. Diese öffnen wir im Block-Editor (Rechtsklick darauf), ziehen die Blöcke auf die Skriptebene und löschen dann auch diese Methode.

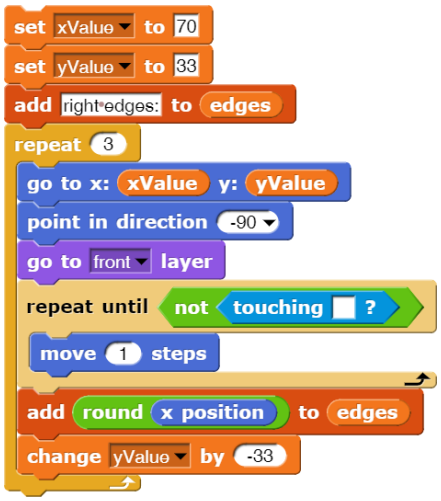
Wie unterscheiden wir nun die Formen der Schilder?

Dafür kann man sich sehr unterschiedliche Verfahren einfallen lassen. Wir versuchen es einmal damit: Wie bestimmen in drei Höhen die horizontalen Grenzen der Schilder und danach an drei Positionen die vertikalen. Dann sehen wir uns die Ergebnisse an.

Zuerst die linken Ränder …

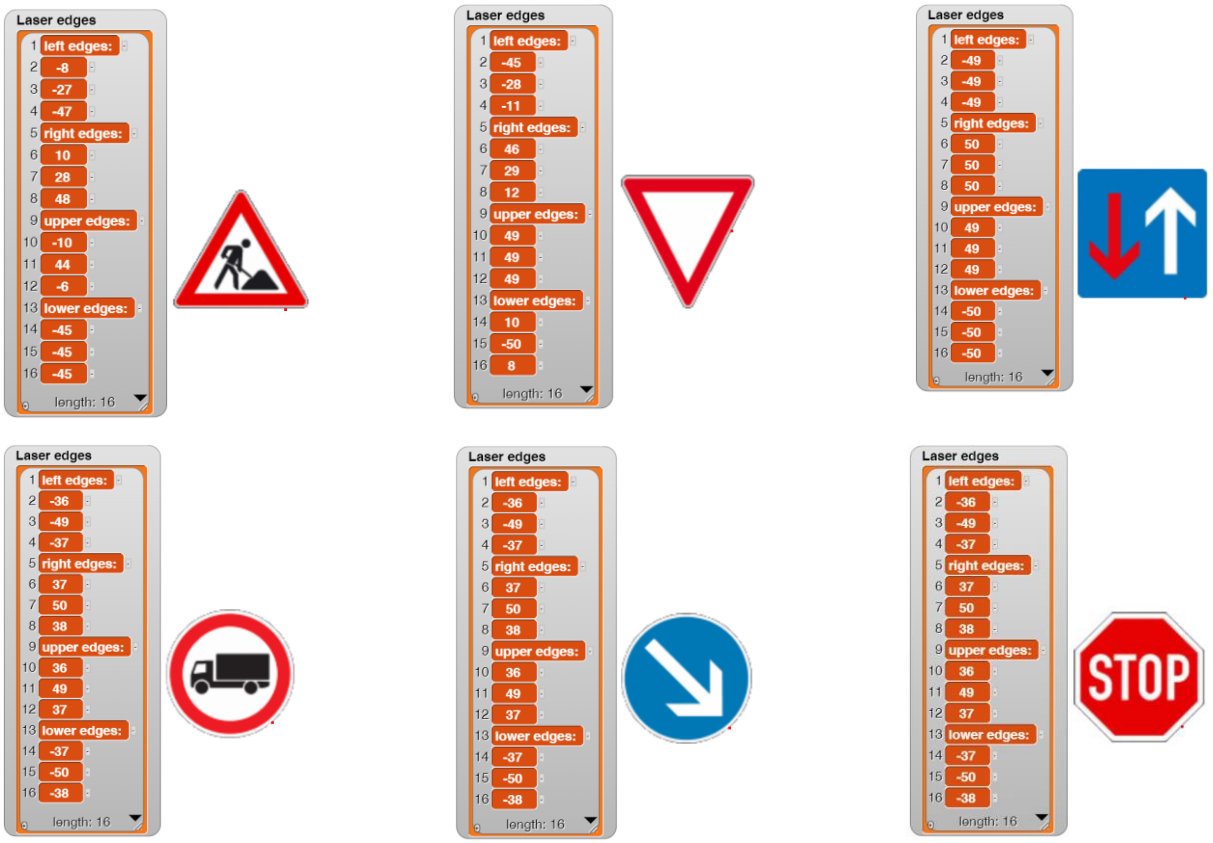


dann die rechten …

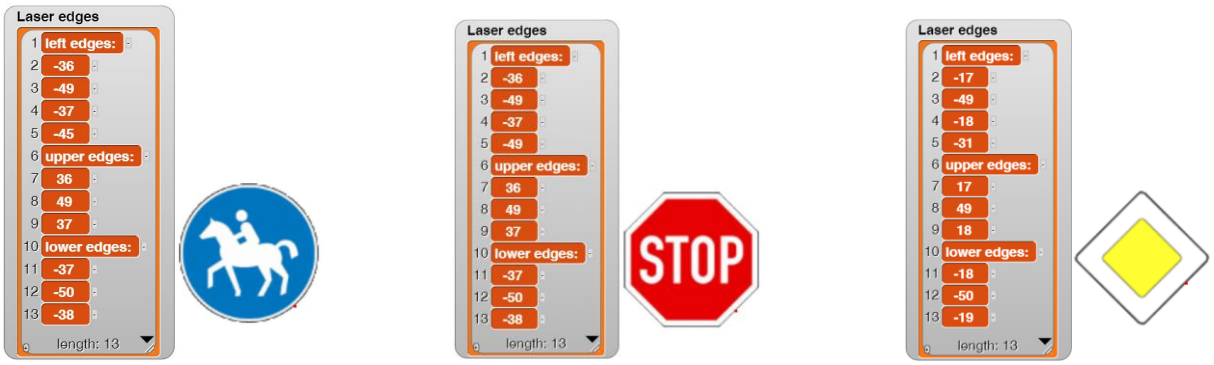


… und entsprechend die oberen und unteren.

Die vier Skripte fügen wir aneinander und verpacken sie in einer Methode *determine edges*. Wir erhalten z. B. die folgenden Ergebnisse.



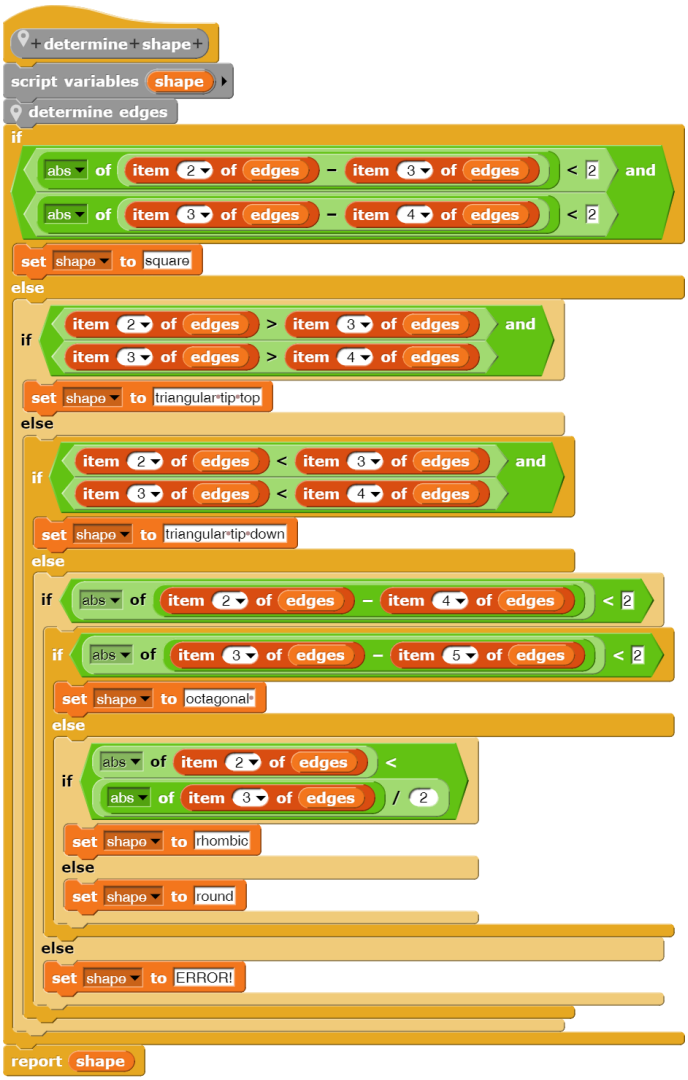
Das sieht doch schon mal gut aus – bis auf das Stop-Schild. Dessen Ränder ähneln verdächtig einem runden Schild; bei denen müssen wir uns noch etwas einfallen lassen. Vielleicht einen 13. „Schnitt“ an geeigneter (hier: vierter, in der Liste: fünfter) Stelle? Dafür können wir die rechten Ränder weglassen, denn die Schilder sind offensichtlich symmetrisch. Machen wir das, dann erhalten wir für die „runden“ Kandidaten:



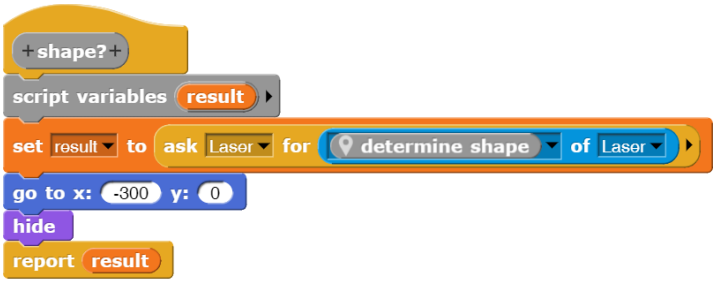
Der 5. Listeneintrag enthält jeweils den Wert für die Höhe 19 – und damit einen messbaren Unterschied.

Zur Auswertung unserer Ergebnisse schreiben wir einen Block *determine shape*. Es soll sich dabei um einen Reporter-Block handeln, der einen Wert – die Form – ermittelt und zurückgibt.

Bei rechteckigen Schildern sollten die Einträge 2, 3 und der 4. Eintrag etwa gleich sein, bei den dreieckigen Schildern wachsen oder fallen die Werte. Vermuten wir etwas Rundes (der zweite und vierte Eintrag sollten etwa gleich groß sein), dann handelt es sich um das Achteck des Stop-Schildes, wenn der dritte und der fünfte Eintrag etwa gleich sind. Und um die Raute des Vorfahrtschildes, wenn der zweite Eintrag ziemlich klein ist. Sonst wirklich um ein rundes Schild. Und Fehler können natürlich auch auftreten.

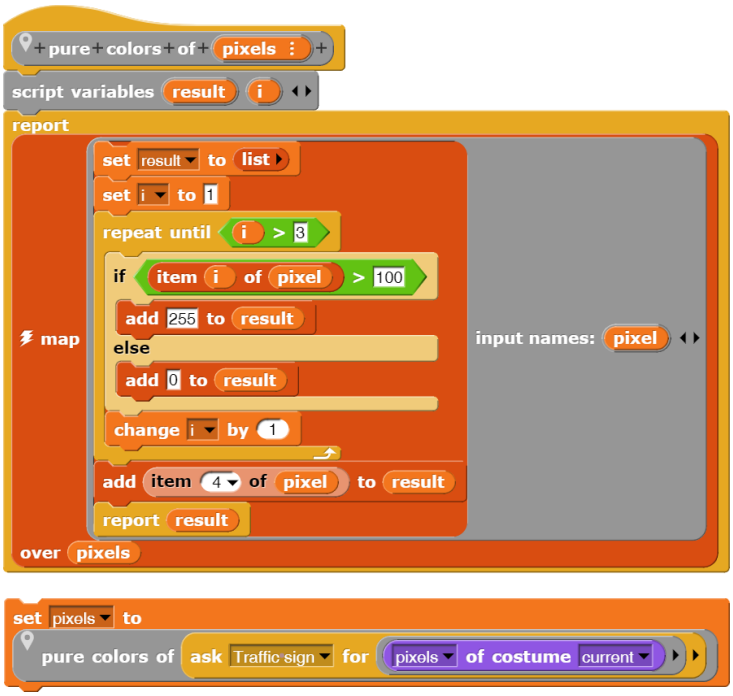


Damit haben wir die Zahl der Möglichkeiten schon ziemlich eingeschränkt, und wir sehen, dass wir – bisher wenigstens – mit den Ergebnissen für den linken Rand auskommen. Wir schreiben eine lokale Methode *shape?*, die die Form des gerade dargestellten Schildes ermittelt. Zusätzlich wird der Laser „in die Heide“ geschickt und versteckt, damit er nicht weiter stört. Seine Arbeit ist erledigt.



Für die weiteren Bedeutungen der Schilder sind die Farben am Rand und im Innern von Bedeutung. Zur endgültigen Bestimmung der Art des Verkehrsschildes wollen wir einfach die Zahl der verschiedenfarbigen Pixel im Schild zählen. Vielleicht reicht das ja. Wir überlassen diese Arbeit einem neuen Objekt namens *Color counter*. Dieses benötigt die Pixelliste des aktuellen Kostüms des Objekts *Traffic sign*. Das bitten wir höflich um die benötigten Daten, die wir in einer lokalen Variablen *pixels* speichern. Dort steht dann eine Liste der drei Farbwerte und der Transparenz der Pixel des aktuellen Kostüms. Da dieses die Maße 100 x 100 hat, bekommen wir 10000 Werte.

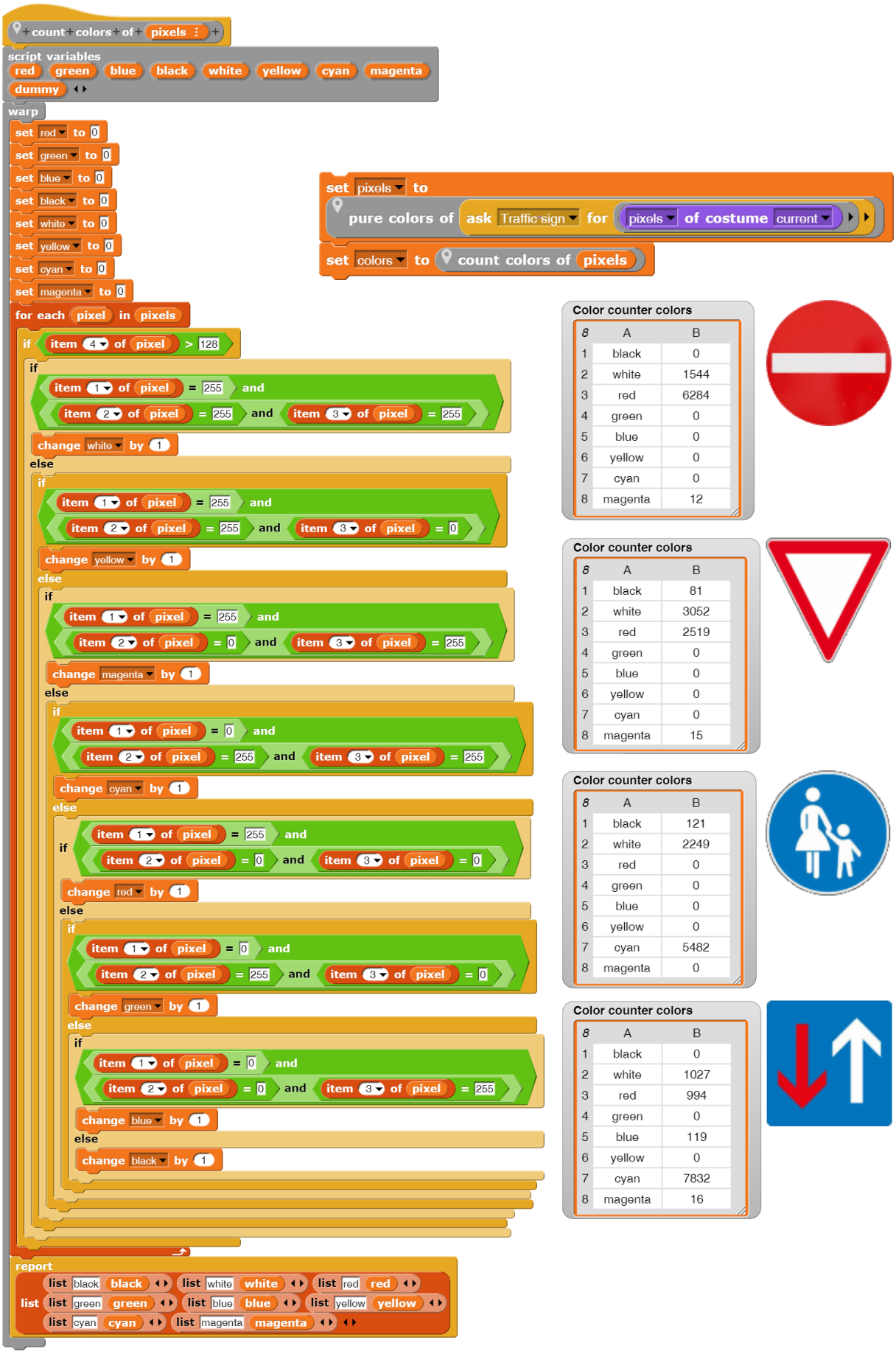
In dieser Liste haben die Pixel außerhalb des eigentlichen Schildes die Transparenz 0, die innerhalb den Wert 255. Die drei RGB-Werte davor stellen keine „reinen“ Farben dar, sondern Mischwerte, die z. B. „überwiegend“ rot sind. Das ändern wir durch eine Methode *pure colors of …*, die Farbwerte über 100 auf 255, die anderen auf 0 setzt. Das geht in der kompilierten Version des *map-over*-Blocks auch bei 10000 Werten sehr schnell.



10000	A	B	C	D
841	237	28	36	255
842	237	28	36	255
843	237	28	36	255
844	237	28	36	255
845	237	28	36	255
846	237	28	36	255
847	237	28	36	255
848	237	28	36	255
849	237	28	36	255
850	237	28	36	255
851	237	28	36	255
852	237	28	36	255
853	237	28	36	255
854	237	28	36	255
855	237	28	36	255
856	237	28	36	255
857	237	28	36	255
858	237	28	36	255
859	237	28	36	255
860	237	28	36	255
861	237	28	36	255

10000	A	B	C	D
841	255	0	0	255
842	255	0	0	255
843	255	0	0	255
844	255	0	0	255
845	255	0	0	255
846	255	0	0	255
847	255	0	0	255
848	255	0	0	255
849	255	0	0	255
850	255	0	0	255
851	255	0	0	255
852	255	0	0	255
853	255	0	0	255
854	255	0	0	255
855	255	0	0	255
856	255	0	0	255
857	255	0	0	255
858	255	0	0	255
859	255	0	0	255
860	255	0	0	255
861	255	0	0	255

Ganz ähnlich lassen wir die „reinen“ Farben im Bild zählen: Wir führen für jede eine eigene Skriptvariable ein, die wir anfangs auf Null setzen. Danach betrachten wir alle Pixel des Schildes, die eine genügend große Transparenz besitzen. Für diese analysieren wir die RGB-Werte und erhöhen den Wert der richtigen Variablen. Zuletzt geben wir eine Liste mit den Ergebnissen zurück, in die wir die Farbbezeichnungen einfügen, damit wir nicht durcheinanderkommen.



Color counter colors

	A	B
1	black	0
2	white	1544
3	red	6284
4	green	0
5	blue	0
6	yellow	0
7	cyan	0
8	magenta	12

Color counter colors

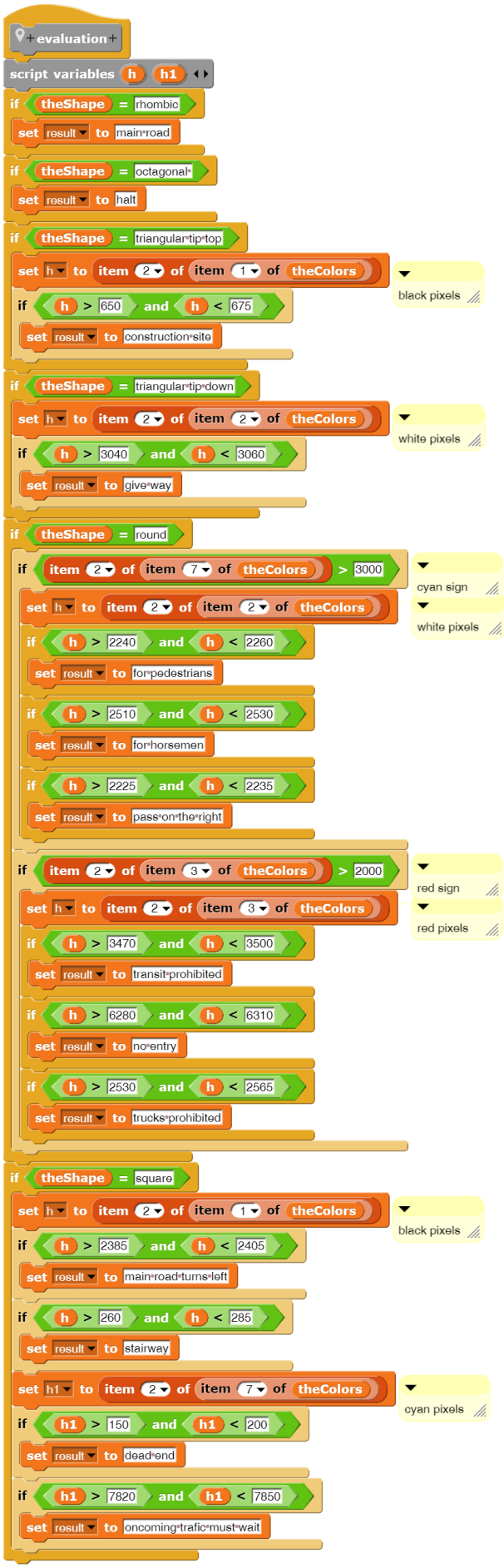
	A	B
1	black	81
2	white	3052
3	red	2519
4	green	0
5	blue	0
6	yellow	0
7	cyan	0
8	magenta	15

Color counter colors

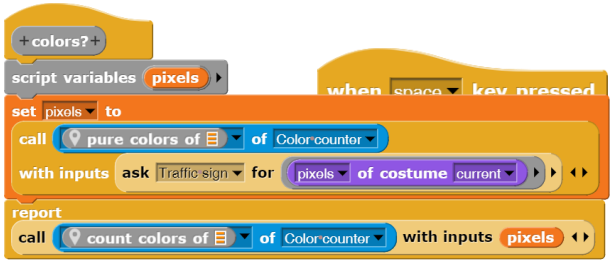
	A	B
1	black	121
2	white	2249
3	red	0
4	green	0
5	blue	0
6	yellow	0
7	cyan	5482
8	magenta	0

Color counter colors

	A	B
1	black	0
2	white	1027
3	red	994
4	green	0
5	blue	119
6	yellow	0
7	cyan	7832
8	magenta	16



Zur einfachen Benutzung der Methoden schreiben wir wieder eine globale Methode *colors?*, die die entsprechenden Operationen veranlasst.



Die Steuerung der Objekte überlassen wir der Bühne. Beim Drücken der Leertaste soll das Verkehrsschild wechseln und beim Anklicken der Grünen Flagge erfolgt die Analyse. Das Bühnen-Objekt fragt die Ergebnisse der anderen ab und wertet deren Daten aus.

Zur Auswertung ziehen wir einerseits die ermittelte Form, andererseits die gezählten Farbwerte heran. In einfacher Form kann das wie nebenstehend beschrieben geschehen.

Die Ergebnisse sind wie gewünscht.

10.3 Projekt: Gesichtserkennung

Altersstufe: *Sekundarstufe II* Material: *Face recognition*

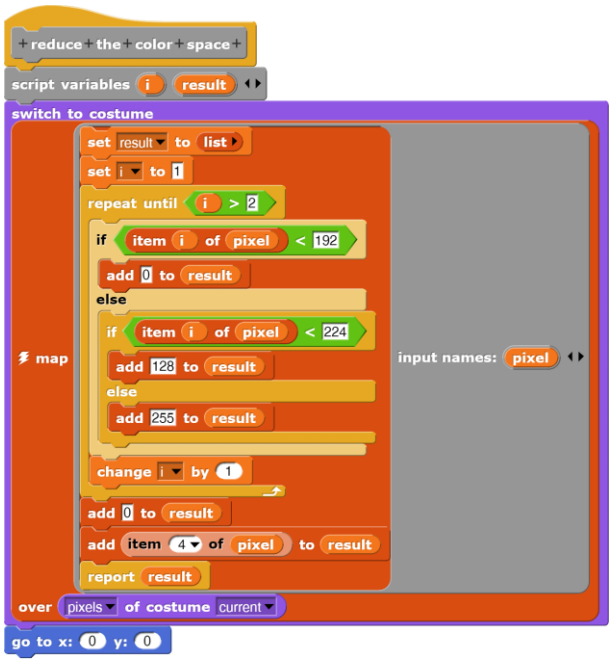
Um die gesellschaftlichen Folgen der Informatiksysteme zu diskutieren, ist die Gesichtserkennung ein gutes Thema. Wir wollen deshalb die schon kennengelernten Möglichkeiten von *Snap!* ansatzweise zu diesem Zweck einsetzen.

Passfotos sind aus guten Gründen stark normiert: die Gesichtshaltung ist vorgeschrieben, Ohren müssen sichtbar sein, … Das erleichtert die Gesichtserkennung erheblich. Wir zeichnen deshalb vier Gesichter, die in etwa diesen Vorschriften entsprechen. Auf diese „Fotos“ wenden wir dann die schon bekannten Verfahren an.

Peter

Paul

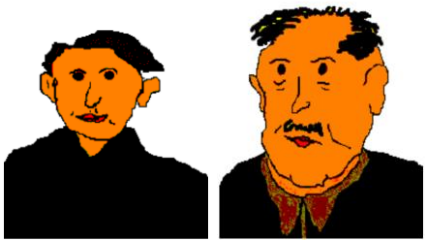
Wir suchen ja das Gesicht, und das ist in diesen vier Fällen in etwa „rosa“. Da die Gesichtsfarben aber trotzdem unterschiedlich sind, führen wir erstmal eine Farbraumreduzierung durch. Geeignete Grenzen der (hier) drei Intervalle finden wir durch Ausprobieren.



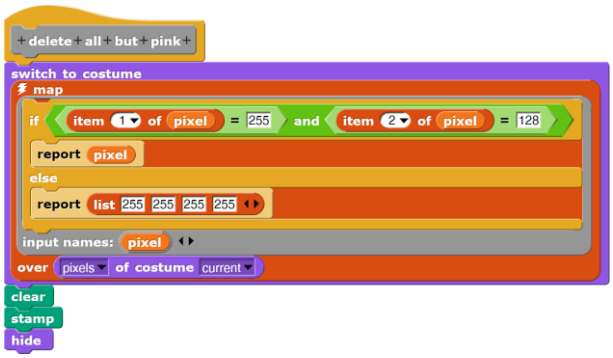
Das Verfahren ist aus der Verkehrsschilderkennung im vorigen Abschnitt. Die Gesichter treten jetzt sehr schön orange hervor – unabhängig davon, wie sie vorher aussahen.

Mary

Hannah



Wenn wir alle Farben außer Orange löschen, bleiben nur noch die Gesichter übrig. Das Ergebnis stempeln wir auf die Bühne und lassen das Passfoto verschwinden. Es hat seine Schuldigkeit getan.



In diesen Gesichtern müssen wir jetzt die Augen, den Mund, die Nase usw. identifizieren. Aus den Verhältnissen der Größen *Augenabstand zu Nasenlänge*, *Mundbreite zu Gesichtshöhe*, … lässt sich dann auf die Person schließen.

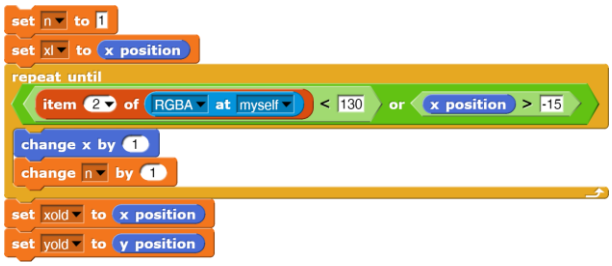
Wie findet man Augen?

Sie stellen „Löcher“ im Gesicht dar, die nicht zu groß und nicht zu klein sein dürfen. Das rechte Auge (aus Sicht der Person) z. B. sollte sich oben-links im Passbild befinden. Dazu müssen wir zuerst einmal auf einzelne Pixel im Bild zugreifen könne. Wir benutzen dazu unser Laser-Sprite, das uns den Farbwert an seiner Position direkt liefert.

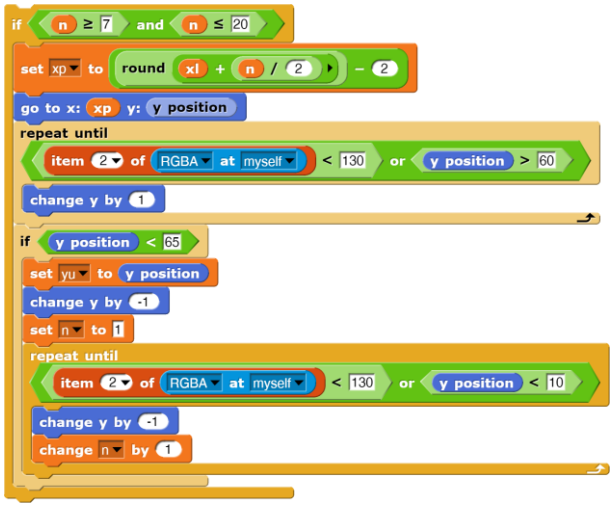
Damit durchsuchen wir den oberen-linken Bildbereich nach einem „Loch“. Wir untersuchen den Bereich -50 < x < -15, 10 < y < 60. Die Werte finden wir durch Ausprobieren. Für die Vergleiche ziehen wir den Grünwert heran, und wir untersuchen den Bereich zeilenweise.

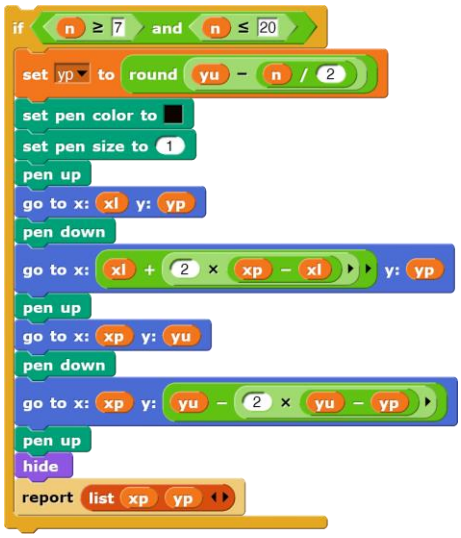
Wir überlaufen einen eventuell vorhandenen orangenen Bereich und stoppen beim ersten weißen Pixel.

Dann merken wir uns den linken x-Wert, zählen die folgenden weißen Pixel nach rechts und merken uns den Endpunkt.

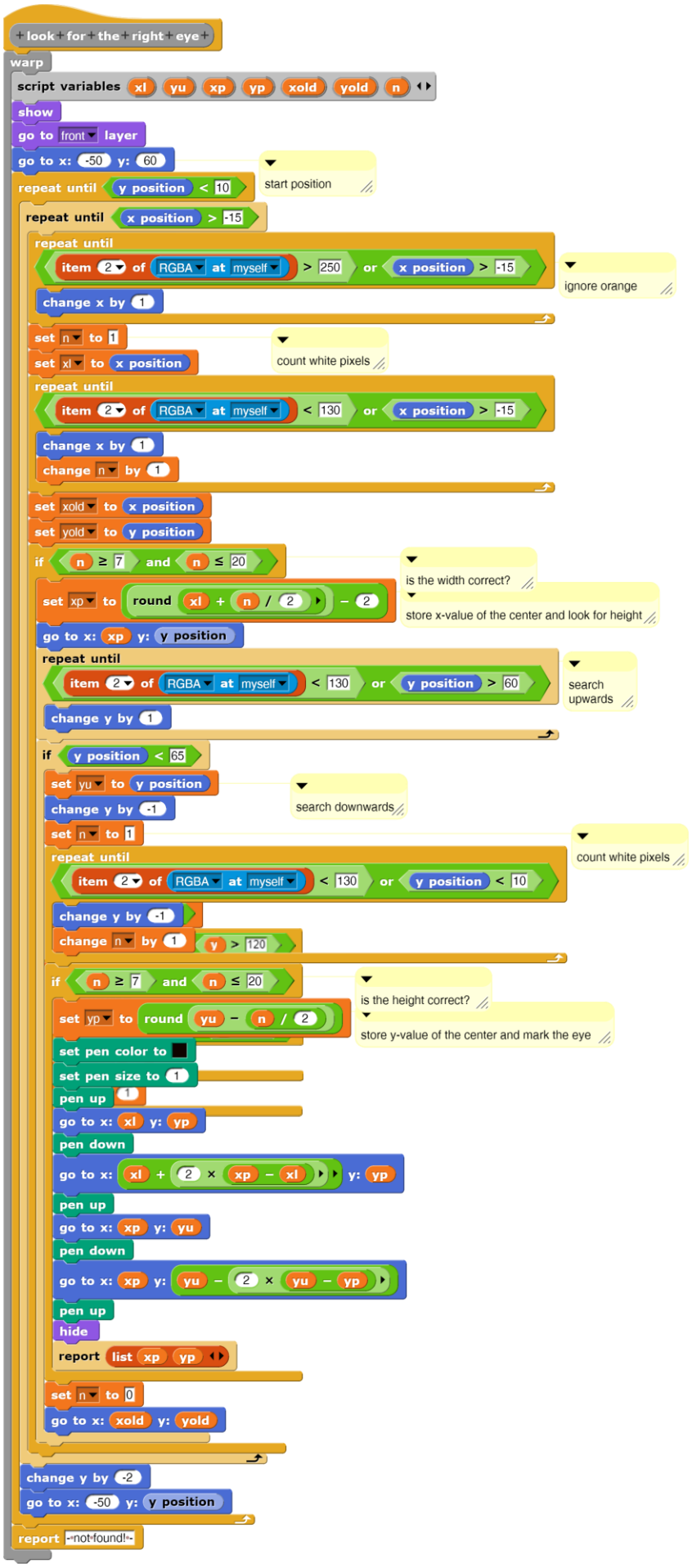


Wenn die Breite einem Auge entspricht, bestimmen wir die Mitte und messen dort die Anzahl der weißen Pixel in der Vertikalen.





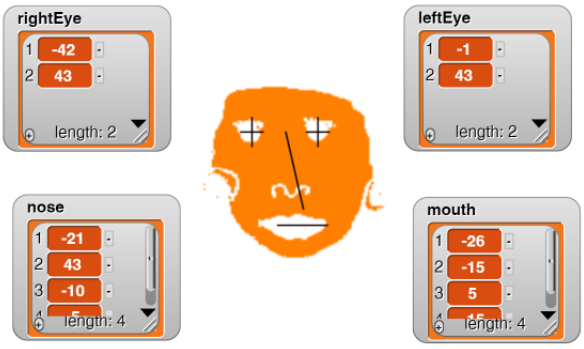
Falls auch das Ergebnis „stimmt“, lassen wir ein Kreuz an der Position zeichnen und geben den Mittelpunkt zurück.

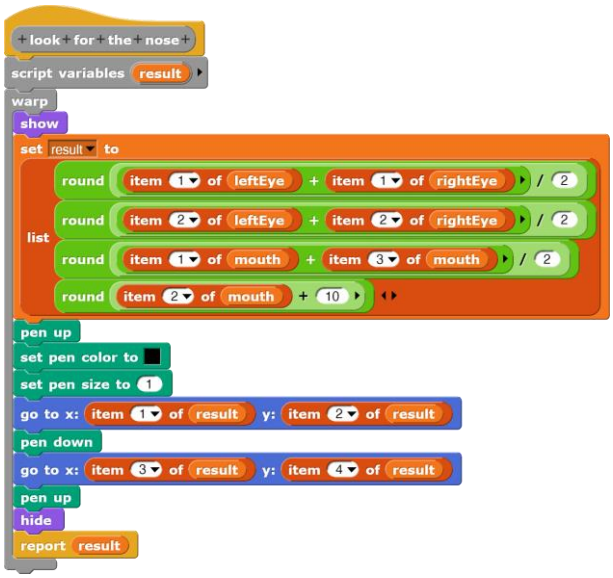


Haben wir noch kein Auge gefunden, dann suchen wir zuerst nach rechts weiter. War auch dort nichts, dann wiederholen wir alles in den nächsten Zeilen.

Nebenstehend das Verfahren im Ganzen.

Das linke Auge finden wir nach dem gleichen Verfahren, und für den Mund bestimmen wir die beiden Mundwinkel. Die Nase zeichnen wir einfach zwischen Augen und Mund.





Aus den ermittelten Werten berechnen wir einige Verhältnisse und speichern sie zusammen mit den Namen in einer Liste *allAttributes*. Durch Vergleich mit den aktuell ermittelten Werten kann dann die gesuchte Person leicht identifiziert werden.

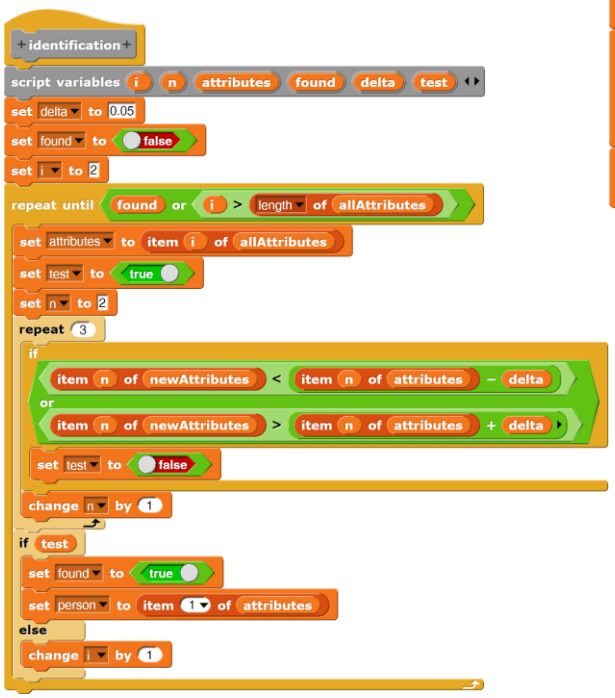
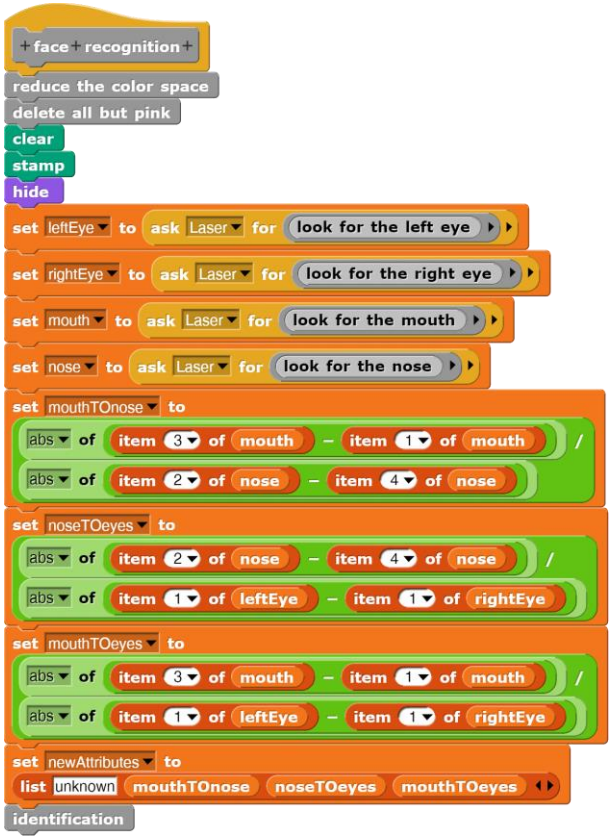


Table view

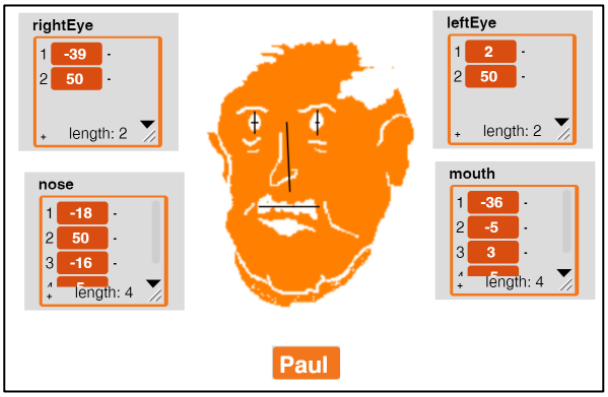
5	A	B	C	D
1	Name	Mouth : Nose	Nose : Eye	Mouth : Eye
2	Mary	0.646	1.171	0.756
3	Hannah	0.707	0.837	0.591
4	Peter	0.962	0.929	0.893
5	Paul	0.867	1.098	0.951

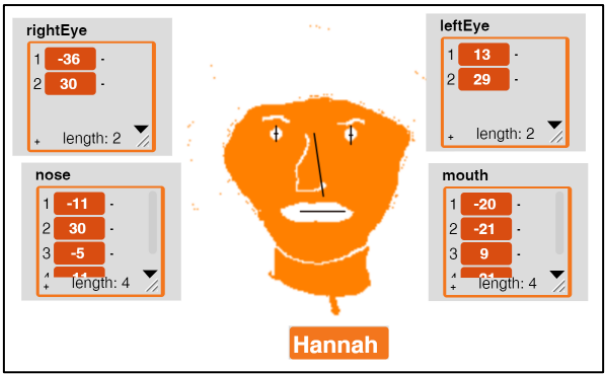
OK

Das Gesamtproblem lässt sich durch Zusammenstellung der Teilprobleme lösen. Wir gehen davon aus, dass sich das Bild der zu identifizierenden Person auf dem Bildschirm befindet. Dieses wird transformiert, auf die Bühne gestempelt und die Änderungen werden angezeigt.



Die vier Personen werden sicher erkannt.





10.4 Aufgaben

1. a: Informieren Sie sich über die Berechnung der **Prüfziffer** im EAN-8-Code. Testen Sie anhand einiger Beispiele, ob Sie das Verfahren verstanden haben.

   b: Lassen Sie beim Barcodescanner nach jedem Lesevorgang überprüfen, ob die Prüfziffer den richtigen Wert hat.

   c: Erweitern Sie den Barcodescanner um weitere Möglichkeiten: Codes können auch „rückwärts“ gelesen werden, und es gibt auch längere Codes, z. B. EAN-13.

   d: Entnehmen Sie gelesenen Barcodes die Hersteller- und die Produktnummer. Geben Sie anhand entsprechender Daten die Ergebnisse im Klartext an: *„Honig vom Bienenhof“*, …

2. Entwickeln Sie einen **Barcode-Generator**. Ihm wird eine Zahlenfolge übergeben. Aus dieser berechnet er die Prüfziffer und druckt den Barcode. Das kann z. B. mithilfe entsprechender Kostüme geschehen, die an den richtigen Stellen mit dem *stamp*-Block aus der *Pen*-Palette auf die Bühne gedruckt werden.

3. Lassen Sie **ausländische Verkehrsschilder** erkennen. Stellen Sie anhand der Schilder fest, wo ein Bild aufgenommen wurde.

4. Ein **Geschwindigkeitswarner** soll in einem Auto anhand wechselnder Verkehrsschilder feststellen, ob die Höchstgeschwindigkeit überschritten wurde.

5. Deutsche **KFZ-Kennzeichen** enthalten einen Zeichensatz, der für die Bilderkennung sehr geeignet ist (einheitliche Zeichenbreite, …). Entwickeln Sie ein Verfahren, das KFZ-Kennzeichen erkennt. Diskutieren Sie die Konsequenzen.

6. **Gesichtserkennung** findet man heute bei der Anmeldung an ein Computersystem, in Kameras und Smartphones, in Sozialen Netzwerken, ... Informieren Sie sich über weitere Anwendungen und diskutieren Sie ihre Ergebnisse.

7. In einigen Staaten wird ein System von **Social Credits** eingeführt oder die Einführung wird diskutiert. Informieren Sie sich über das System und diskutieren sie die Konsequenzen in Verbindung einer umfangreichen Videoüberwachung.

11 Klänge

Ähnlich wie bei bewegter Grafik ist es etwas schwierig, den Umgang mit Klängen nur zu beschreiben. Es werden deshalb hier nur die verschiedenen Möglichkeiten dargestellt – mit der dringenden Empfehlung, die „Codeschnipsel“ dann auch auszuprobieren und damit zu experimentieren.

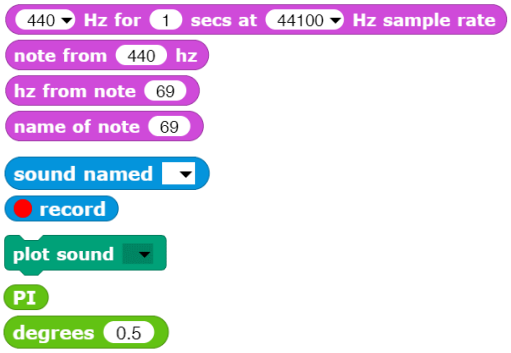
11.1 Klänge finden

Zuerst einmal benötigen wir einen Klang im *WAV*-Format. Dazu können wir den entweder über das Dateimenü (*File → Sounds…*) importieren …

… oder wie gewohnt „von außen“ in das *Snap!*-Fenster ziehen …

… oder einfach selbst aufnehmen. Das kann – bei kurzen Aufnahmen – direkt mithilfe des *Snap!*-Tonrecorders auf der *Sounds*-Seite geschehen. Für längere Aufnahmen sollte man eines der gängigen Tools benutzen.

Für die weitere Bearbeitung laden wir die Bibliothek *Audio Comp* aus dem Dateimenü. Damit stehen uns insgesamt die nebenstehenden Blöcke aus den *Sound-*, *Sensing-*, *Pen-* und *Operators*-Menüs zur Verfügung.



Im Folgenden arbeiten wir mit der Datei *Tontest.wav*, die wir auf einem der beschriebenen Wege erzeugt haben.

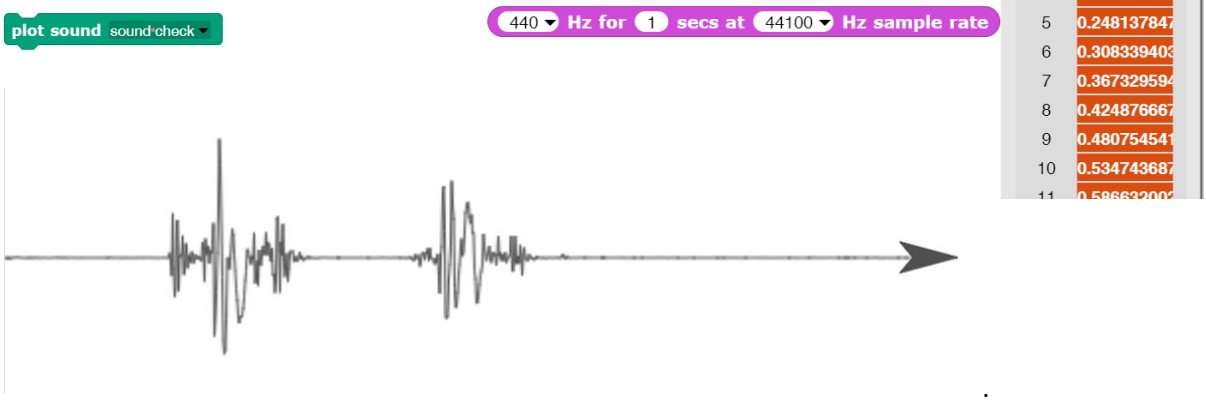
11.2 Klänge verarbeiten

Ist ein Klang auf der *Sounds*-Seite vorhanden, dann kann er in den entsprechenden Blöcken angezeigt werden. Am einfachsten probiert man das im Block zum Abspielen von Sounds.

Für die weitere Verarbeitung benötigen wir einen Repräsentanten unseres Sounds. Dafür ist der Block *sound named <sound-name>* gedacht. Editiert man den, dann hat man gleich ein kleines Beispiel zur Nutzung der Sound-Blöcke gefunden.[59]

Der *of*-Block für Sounds bietet Zugriff auf weitere Eigenschaften von Sounds. Insbesondere lassen sich seine *samples*[60] als Liste ermitteln. Die werden benötigt, wenn ein Sound aktiv bearbeitet werden soll. Wir können z. B. die Abspielgeschwindigkeit des Sounds durch Änderungen der Samplerate beeinflussen. Der *Hz for …*-Block erzeugt Samples mit den anzugebenden Eigenschaften, z. B. „reine Töne“.

Interessant ist die Visualisierung der Klänge. Mithilfe des *plot <sound>* -Blocks erhalten wir eine Grafik des Samples auf der Bühne.



[59] Das Entsprechende gilt für (fast) alle weiteren Sound-Blöcke. Editiert man sie, dann findet man Beispiele z. B. zum Einsatz von JavaScript.

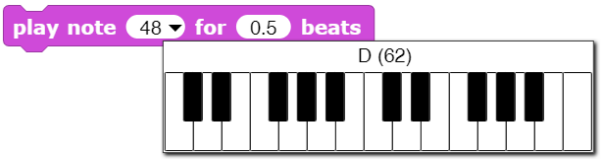
[60] https://de.wikipedia.org/wiki/Abtastrate

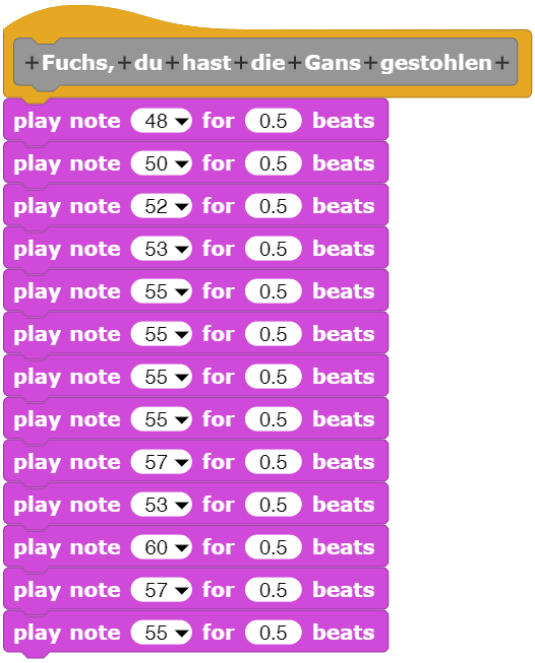
11.3 Musik machen mit Jens Mönig[61]

Altersstufe: *Sekundarstufe II* Material: *Music*

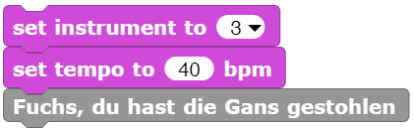
Ein Sample besteht aus einer Liste von Zahlen, Stereosounds aus einer zweielementigen Liste von Samples (s. o.). Folglich lassen sich Sounds mit den üblichen Listenoperationen manipulieren, etwa umkehren, im Wert ändern, …

Songs lassen sich aber auch aus Noten zusammensetzen, sogar ganz komfortabel. Die Wahl der Note erfolgt auf einer Klaviatur (Klavier-Tastatur), die man erhält, wenn man auf dem *play note…for…beats*- Block den kleinen Pfeil nach unten für das Drop-down-Menü anklickt. Daraus lassen sich schnell Lieder zusammensetzen …





… und auf verschiedenen Instrumenten und in verschiedenen Tempi abspielen.



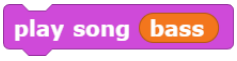
[61] Nach dem Beispiel „music“ von Jens Mönig

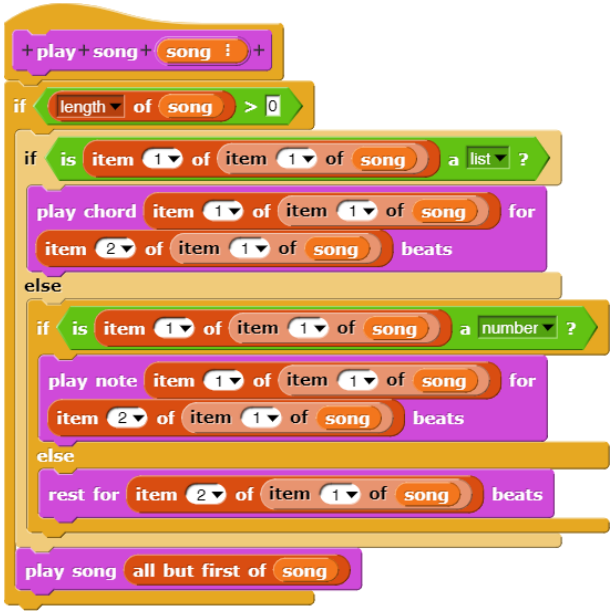
Spielt man mehrere Noten parallel, dann entstehen Akkorde ...

… und aus diesen Lieder, …

… die mithilfe geeigneter Liste aus Paaren von (Note, Dauer) …

… abgespielt und variiert werden können.





zwei Grundakkorde vorgeben

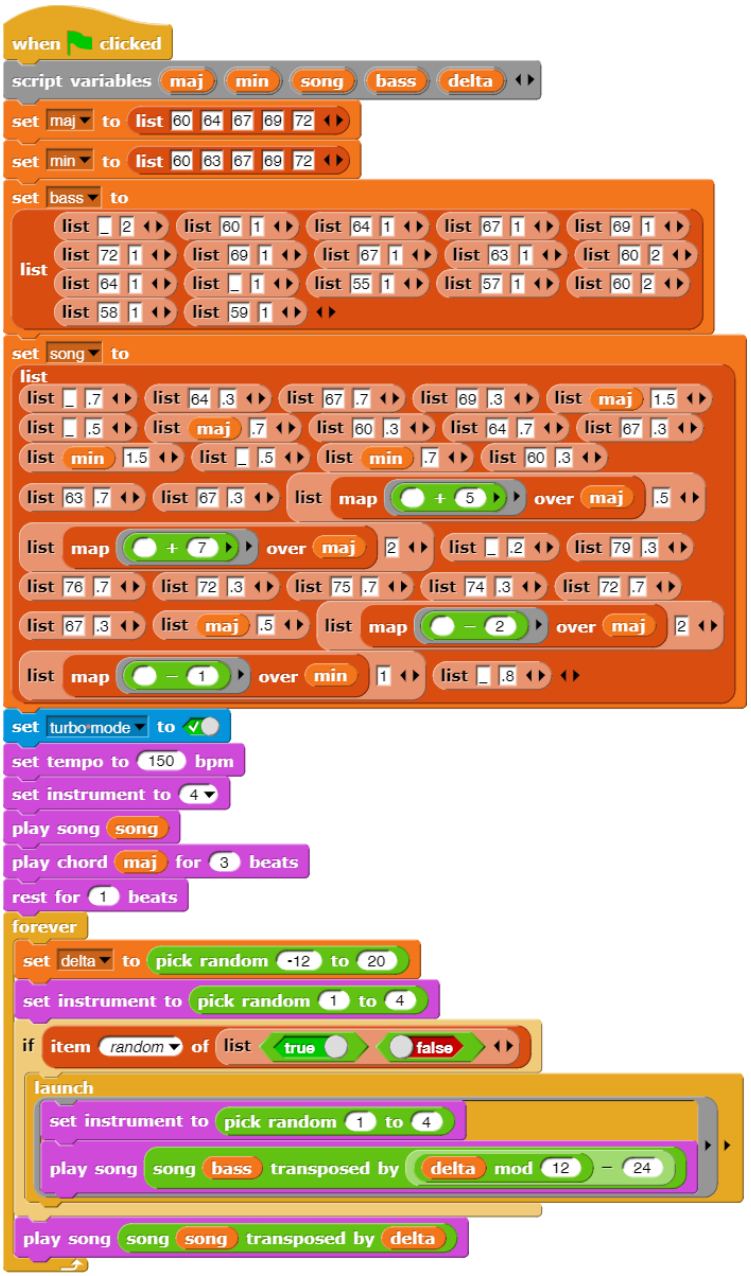
Bassbegleitung und Lied durch Listen aus Ton/Dauer-Paaren beschreiben

ein paar Voreinstellungen treffen

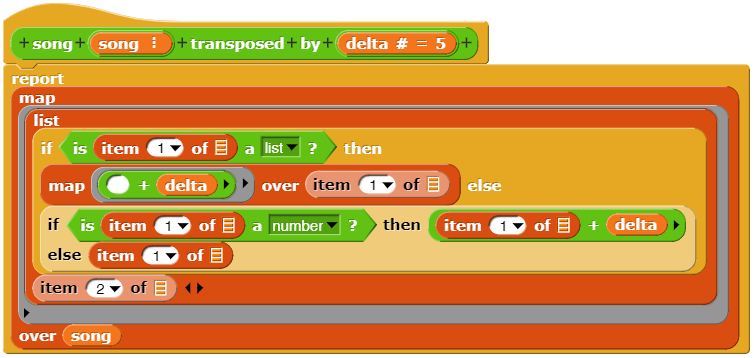
den Song spielen, mit Akkord abschließen und kurze Pause

und jetzt mit Variationen das Lied und die Bassbegleitung immer wieder abspielen

beide spielen parallel wegen des *launch*-Blocks

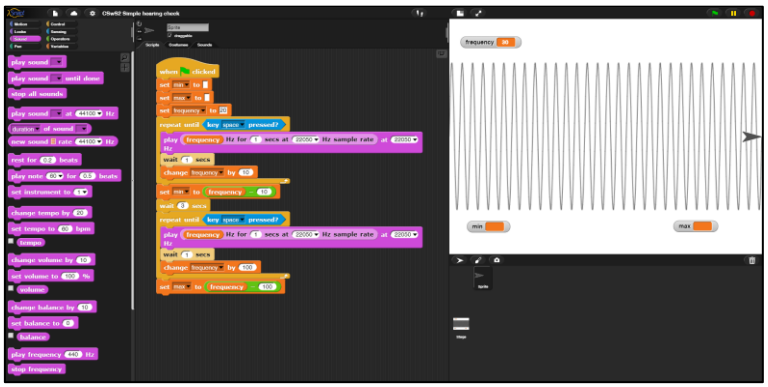


einen Song transponieren



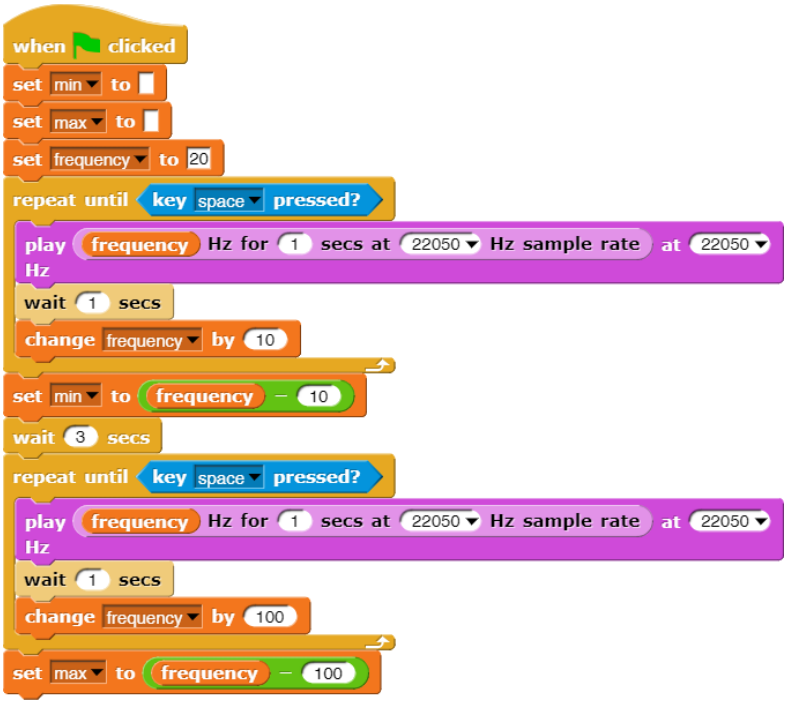
11.4 Projekt: Hörtest

Altersstufe: *Sekundarstufe I* Material: *Hearing volume test*



Bei einem Hörtest wird die Hörfähigkeit bei verschiedenen Frequenzen oder verschiedenen Lautstärken getestet. In diesem Fall spielen wir solange Töne steigender Frequenz ab, bis der (oder die) Proband(in) etwas hört. Dann drückt er (oder sie) die Leertaste. Diese Frequenz *min* wird vermerkt. Danach wird die Frequenz solange erhöht, bis nichts mehr gehört wird. Auch diese Frequenz wird gespeichert. In der derzeitigen *Snap!*-Version müssen dafür die *JavaScript*-Extensions im *Settings*-Menü freigeschaltet werden.

Achten Sie darauf, dass die Lautstärke nicht zu groß werden kann!

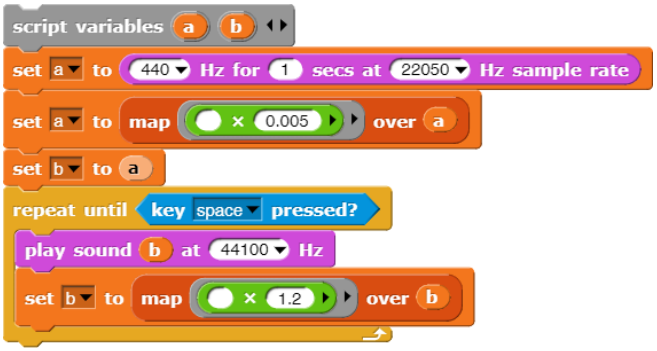


11.5 Aufgaben:

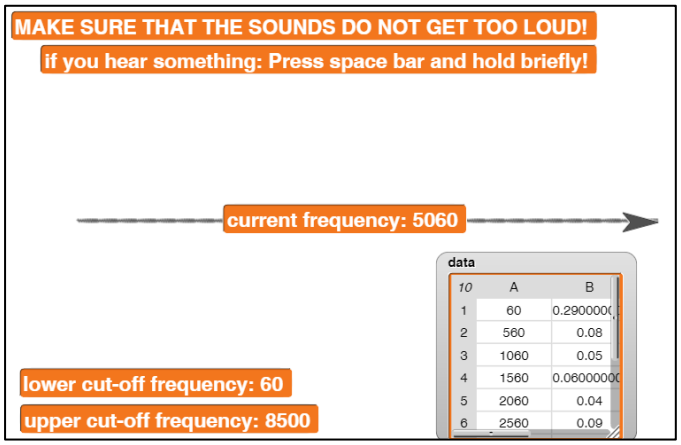
1. Stellen Sie im Hörtest Versuchsbedingungen her, die zu vergleichbaren Ergebnissen führen.

2. Ändern Sie nicht nur die Frequenz, sondern auch die Lautstärke. Da unsere Sounds durch Samples beschrieben werden, lässt sich die Lautstärke durch einfache Multiplikation der Samplewerte ändern. Z. B. wird im folgenden Skript die Lautstärke so lange erhöht, bis die Leertaste gedrückt wird.

   Achtung: Die Lautstärke darf nicht zu groß werden!



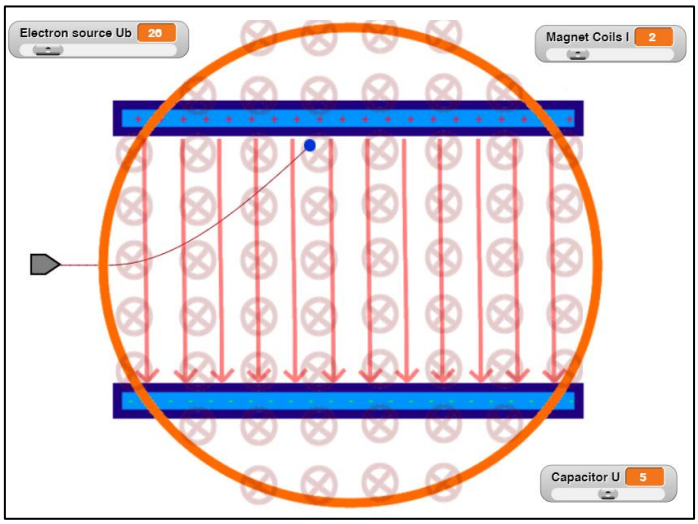
3. Messen Sie die Grenzfrequenzen und die pro Frequenz erforderliche Lautstärke, die zum Hören erforderlich ist. Erstellen Sie daraus ein Diagramm.



4. Machen Sie eine Exkursion zu einer HNO-Praxis/einer HNO-Klinik. Legen Sie ihre Diagramme vor und lassen Sie sich erklären, ob und was man daraus ablesen kann. Informieren Sie sich über Ursachen möglicher Hörschäden.

12 Projekt: Elektronen in Feldern

Altersstufe: *Sekundarstufe II* Material: *Electrons in fields*

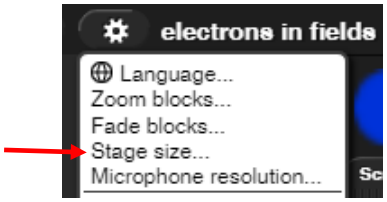


Wir wollen die bisher erworbenen Kenntnisse benutzen, um ein kleines Projekt aus dem Bereich – na ja – Physik zu realisieren: Elektronen bewegen sich in einer Röhre, in die ein Kondensator eingebaut ist. Diese Röhre wird so ins Innere eines Helmholtz-Spulenpaars gebracht, dass elektrisches und magnetisches Feld senkrecht zueinander stehen. Beide sind halbwegs homogen. Es handelt sich dabei um einen der Standard-Schulversuche der Oberstufe. Alle Komponenten können unabhängig voneinander in unterschiedlichen Gruppen entwickelt werden, und das auf sehr unterschiedliche Weise. Nur die Physik bleibt gleich. So ist das nun mal mit der Physik. 😉

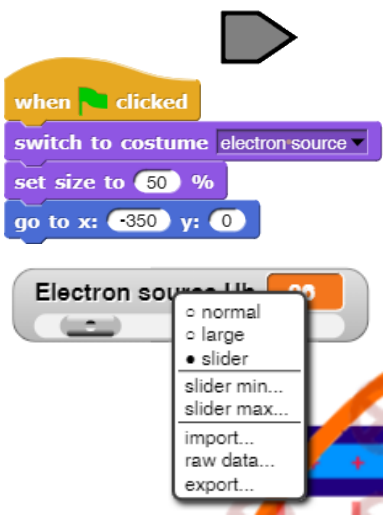
12.1 Die Elektronenquelle und der Versuchsaufbau

Da es sich um ein Standardexperiment handelt, sollten sich die benötigten Geräte in der Physiksammlung finden lassen. Es bietet sich also an, den Versuch schön übersichtlich aufzubauen, ihn zu fotografieren und die Teilgeräte aus den Bildern so zu extrahieren, dass sie sich im Projekt verwenden lassen. Hier im Skript wurden stattdessen nur einfache Zeichnungen gemacht. Wir benötigen Bilder des Kondensators, der Spulen, der Elektronenquelle und – zur Veranschaulichung – der erzeugten Felder.

Zuerst einmal vergrößern wir die Bühne von *Snap!* auf 800 x 600 Pixel. Dafür gibt es einen Menüpunkt im Einstellungsmenü von *Snap!*. Dann zeichnen wir ein einfaches Bild einer Elektronenquelle und importieren es als Kostüm des aktuellen Sprites.



Nach Programmstart mit der grünen Flagge wird unsere Elektronenquelle im richtigen Gewand an ihren Platz geschickt. Bei Bedarf können wir sie im Experiment auch an eine andere Stelle schieben. Das Gerät hat nur eine kennzeichnende Eigenschaft: die momentane Beschleunigungsspannung der ausgesendeten Elektronen. Dafür wird eine lokale Variable *Ub* erzeugt und auf der Stage angezeigt. Im Kontextmenü dieser Anzeige kann *slider* ausgewählt und der Minimal- und Maximalwert eingestellt werden. Mit dem Schieberegler wird der Variablenwert dann im laufenden Programm zwischen diesen Werten verändert. Wir wählen einen Bereich zwischen 0 und 250 (Volt).

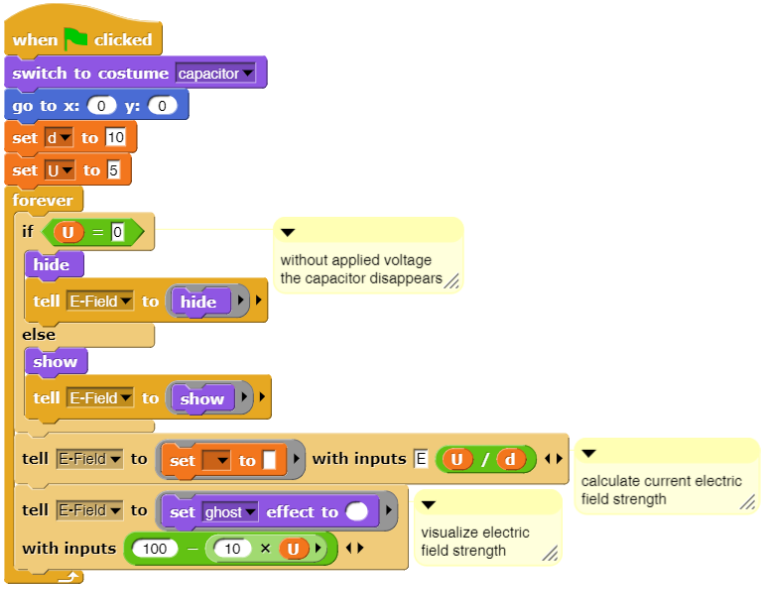


12.2 Der Kondensator und das elektrische Feld

Der Kondensator in der Röhre hat einen Plattenabstand *d*, den wir fest so einstellen, dass sich später eine brauchbare Elektronenbewegung ergibt. Nachdem er auch seinen Platz gefunden hat, läuft er durchgehend bis zum Programmabbruch. Falls wir die anliegende Spannung *U* auf Null setzen, soll er verschwinden, damit wir auch Bewegungen nur im Magnetfeld untersuchen können – da würde er nur stören. Für *U* und *d* richten wir lokale Variable ein. Danach teilt er dem elektrischen Feld *E-Feld* mit, welchen Wert es gerade hat. Das geschieht, indem er im Kontext des E-Feldes den Wert von dessen lokaler Variablen *E* neu mit dem Wert *U/d* setzt.

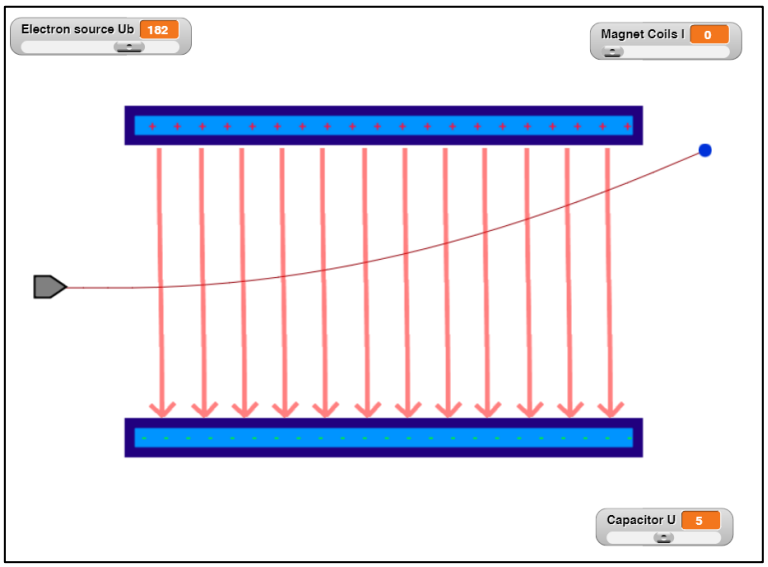
Es gilt nämlich: $E = \frac{U}{d}$

Dabei ist wichtig, dass die Wertefelder in *set <variable> to <wert>* wirklich leer sind, damit sie durch die angegebene Größen ersetzt werden können!



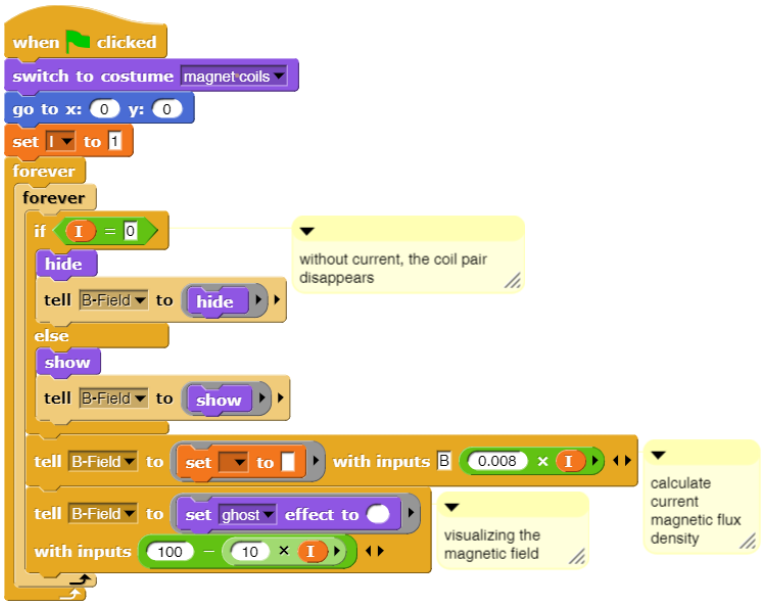
Danach setzt er auf die gleiche Weise den *ghost-effect* des elektrischen Feldes, also seine Transparenz, auf einen Wert, der von der anliegenden Spannung abhängt. Je kleiner diese ist, desto durchscheinender erscheinen die Pfeile, die das elektrische Feld symbolisieren.

Das elektrische Feld, ein weiteres eigenes Sprite, besteht einfach aus einem Kostüm, das eine Reihe paralleler Pfeile enthält, die zwischen die Kondensatorplatten passen. Es besitzt eine lokale Variable *E*, die vom Kondensator – wie beschrieben – gesetzt wird. Die Spannung des Kondensators lassen wir als Slider-Variable auf der Stage anzeigen.

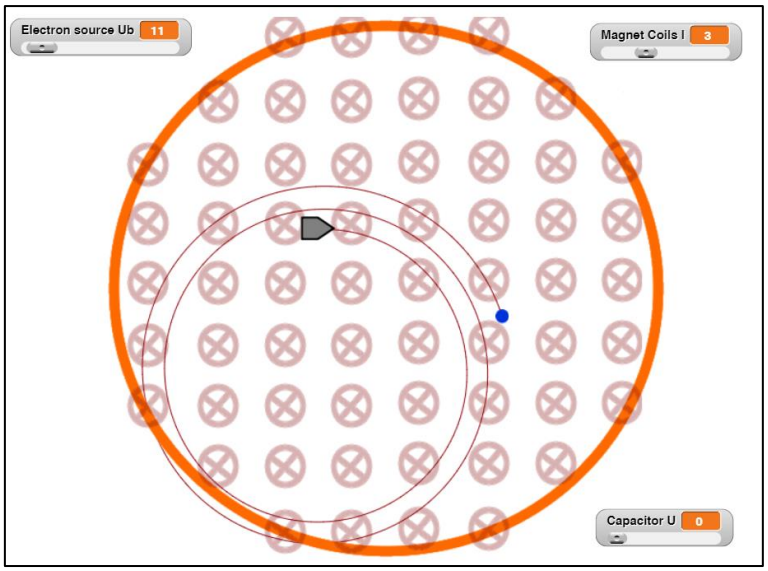


12.3 Die Helmholtzspulen und das magnetische Feld

Das Helmholtzspulenpaar wird durch einen einfachen Kreis auf der Bühne symbolisiert.[62] Es enthält eine lokale Variable *B*, die magnetische Flussdichte, die sich für handelsübliche Geräte zu $B = 0.008\frac{T}{A} \cdot I$ ergibt, wobei *I* die elektrische Stromstärke durch die Spulen ist. Wir lassen sie als Slider-Variable zwischen 0 und 10 (Ampere) anzeigen. Das ist ganz schön kräftig. Die Spulen teilen ähnlich wie der Kondensator dem magnetischen Feld mit, welchen Wert und welche Transparenz es hat. Das Magnetfeld besteht wie das elektrische Feld nur aus einer Zeichnung.



Schalten wir das elektrische Feld ab und betrachten nur die Elektronenbahn im Magnetfeld, dann erhalten wir zwar eine annähernd kreisförmige Bahn, aber keine geschlossene. Die Spirale ergibt sich durch Rechenungenauigkeiten, weil die berechneten Änderungen viel zu groß sind. Wir müssten viel kleinschrittiger vorgehen. Daran wäre also noch zu arbeiten!



[62] Das kann man wirklich sehr viel schöner machen!

12.4 Die Elektronen

Jetzt kommt der bittere Moment, wo wir der Physik nicht mehr ausweichen können. Sei´s drum. 😉

Auf ein Elektron wirken in der Anordnung zwei Kräfte: die elektrische und die magnetische. Mit der elektrischen ist es ziemlich einfach. Sie wirkt hier nach oben, weil das Elektron negativ geladen ist: $F_{e,y} = e \cdot E$

Die Lorenzkraft $\vec{F}_L = q \cdot \vec{v} \times \vec{B}$ steht senkrecht zur aktuellen Geschwindigkeit des Elektrons und zur Feldrichtung. Wir müssen also mit Vektoren arbeiten. Das Magnetfeld hat nur eine Komponente in z-Richtung, also „in den Bildschirm hinein“, die Geschwindigkeit nur zwei Komponenten in x und y-Richtung „auf dem Bildschirm“. Es gilt also: $\vec{F}_L = e \cdot \begin{pmatrix} v_x \\ v_y \\ 0 \end{pmatrix} \times \begin{pmatrix} 0 \\ 0 \\ B \end{pmatrix} = e \cdot \begin{pmatrix} v_y \cdot B \\ -v_x \cdot B \\ 0 \end{pmatrix}$

Zusammengefasst: $\vec{F}_{gesamt} = e \cdot \begin{pmatrix} v_y \cdot B \\ E - v_x \cdot B \\ 0 \end{pmatrix}$, und da gilt: $\vec{F} = m \cdot \vec{a}$

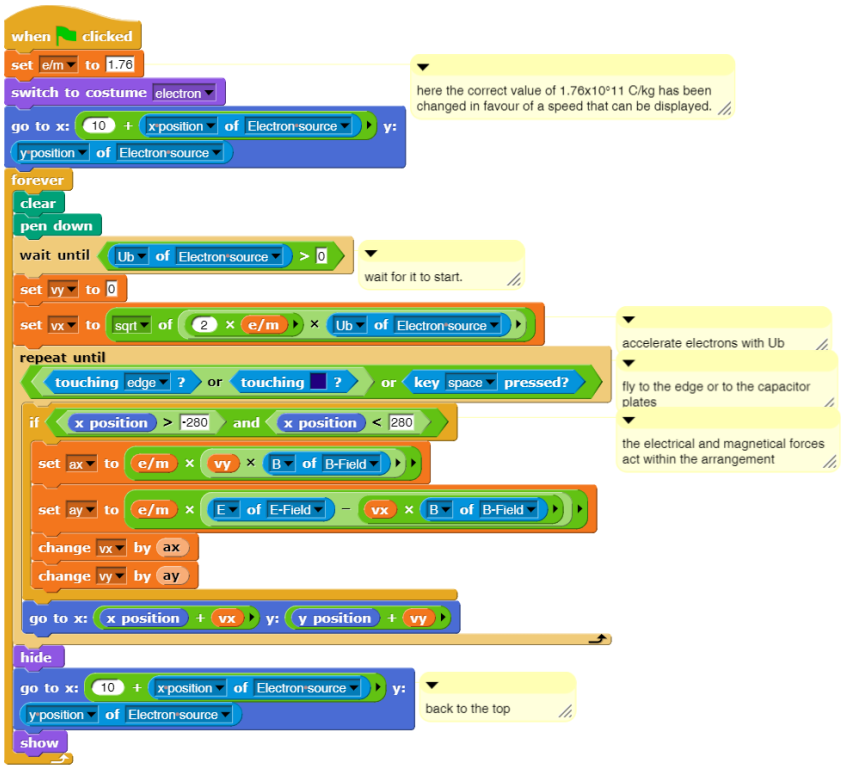
erhalten wir für die Beschleunigungen in den beiden Richtungen:

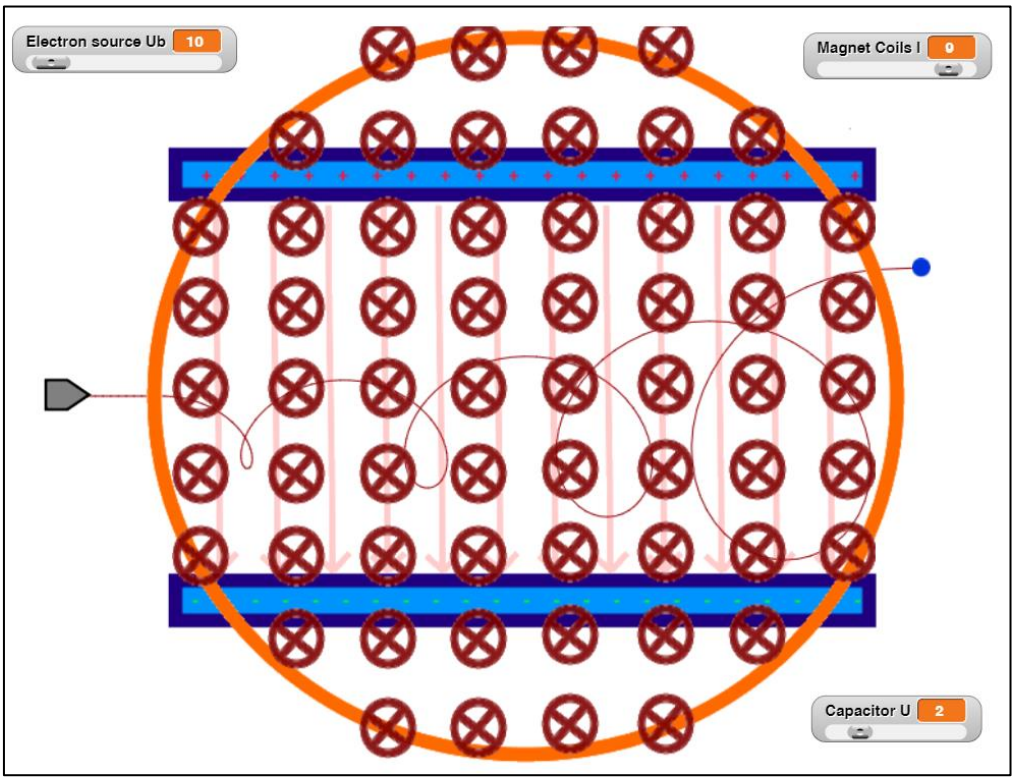
$$a_x = \frac{e}{m} \cdot v_y \cdot B \quad \text{und} \quad a_y = \frac{e}{m} \cdot (E - v_x \cdot B)$$

mit den passenden Vorzeichen zu den Koordinatenrichtungen von *Snap!*. Diese Beschleunigungen ändern die Geschwindigkeitskomponenten und diese wiederum die Position des Elektrons. Das war´s schon.

Wir können diese Ergebnisse direkt in das Skript des Elektrons übernehmen. Die Naturkonstante *e/m* passen wir dazu etwas an, weil „echte“ Elektronen bedeutend schneller sind als unsere Bildschirmvertreter. Andere Anpassungen sind nicht erforderlich. Das Elektron benötigt also nur die „zu groß“ gewählte lokalen Variablen *e/m* und die Beschleunigungs- und Geschwindigkeitskomponenten. Damit man die Bahn besser verfolgen kann, wird sie auf die Stage gezeichnet.

Man kann die manchmal erstaunlichen Bewegungen der Teilchen jetzt schön beobachten. Zu fragen ist natürlich, was davon stimmt und was auf numerische Effekte zurückzuführen ist. Projekte enden eben nie, sie geben Anstöße zu weiteren Fragen!



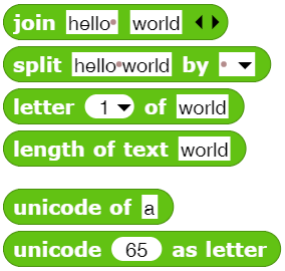


13 Texte und Verwandtes

13.1 Operationen auf Zeichenketten

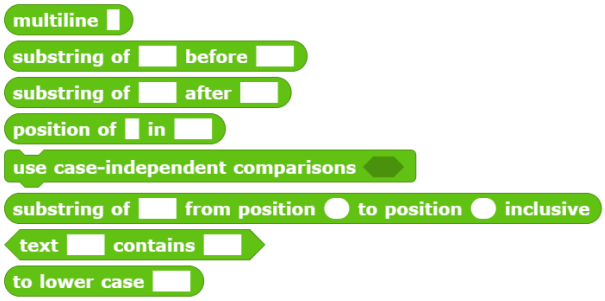
Altersstufe: *Sekundarstufe I* Material: *Stringoperations*

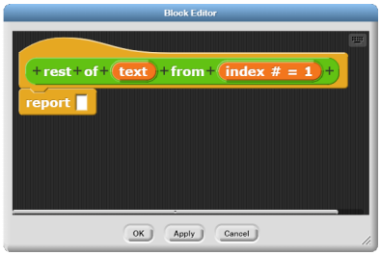
Snap! enthält wie seine Vorgänger einen auf das Wesentliche reduzierten Satz von Methoden, die mit Zeichenketten (Strings) arbeiten. Dazu gehören



- *join <string1> <string2>…* : der Operator für die Konkatenation (Verkettung) von mehreren Zeichenketten. Das Resultat ist eine neue Zeichenkette. Der Operator kann mithilfe der Pfeiltasten um weitere Argumente erweitert werden.
- *split <string> by <zeichen>* : der Operator für die Aufspaltung einer Zeichenkette in eine Liste. Die Auftrennungen erfolgen an den angegebenen Zeichen, typischerweise den Leerzeichen.
- *letter <n> of <string>* : liefert das n-te Zeichen einer Zeichenkette.
- *length of text <string>* : liefert die Länge einer Zeichenkette.
- *unicode of <zeichen>* : liefert den Unicode eines Zeichens.
- *unicode <n> as letter* : liefert das n-te Unicode-Zeichen.

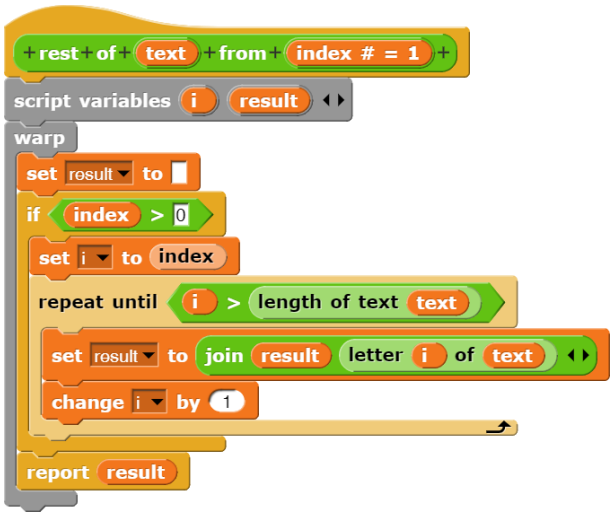
Weitere Zeichenkettenoperationen befinden sich in den Bibliotheken. Sie können über das Datei-Menü importiert werden. Die neuen Blocks befinden sich dann unter dem *Make a block*-Button in der *Operators*-Palette.



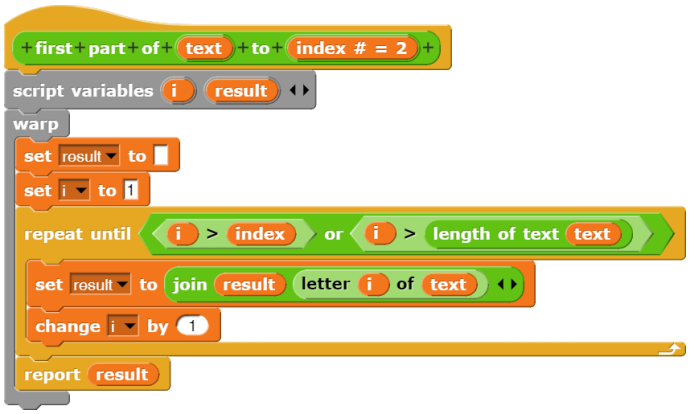


Wir wollen hier einen anderen Weg gehen, indem wir eventuell benötigte Methoden aus den Grundoperationen aufbauen. Als erstes wollen wir eine Methode *rest of <text> from <index>* schreiben, die den Rest einer Zeichenkette ab einem bestimmten Index zurückgibt. Wir erstellen also einen neuen Block, den wir diesmal der *Operators*-Palette zuordnen, damit er bei den String-Operatoren schön grün erscheint. Da es sich um eine Funktion handelt, klicken wir „*Reporter*“ an, und weil natürlich auch andere von unserer Arbeit profitieren sollen, belassen wir es bei „*for all sprites*“. Die Parameter können wir, wie schon mehrfach beschrieben, an den +-Zeichen zwischen den Worten des Methodenkopfes einfügen. Wir typisieren sie als *Text* bzw. *Number* und geben beim Parameter *index* den Default-Wert *1* vor. Beides wird im Methoden-kopf als *index # = 1* angezeigt.

Im Skript kopieren wir alle Zeichen des Textes ab dem Indexwert in eine Stringvariable *result*. Diese geben wir mithilfe des *report*-Blocks als Funktionsergebnis zurück. Um das Ganze schön schnell zu machen, verpacken wir es in einen *warp*-Block.

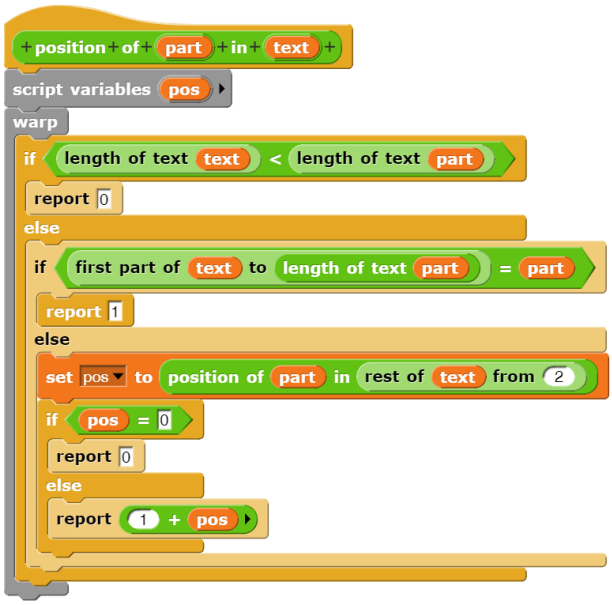


Auf ganz ähnliche Weise liefert die Funktion *first part of <text> to <index>* den Anfang einer Zeichenkette.

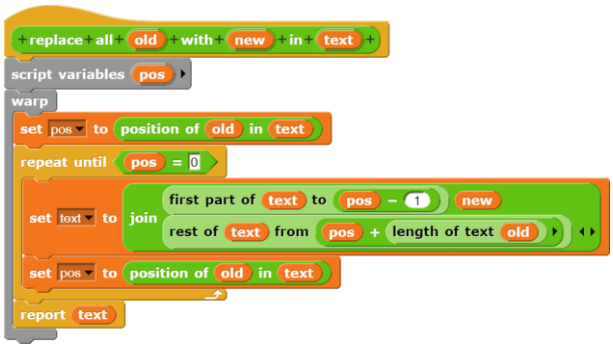


Mit beiden Funktionen ist es leicht, einen Ausschnitt aus einer Zeichenkette zu erhalten.

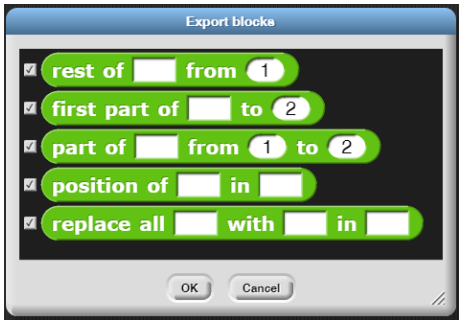
Und die Position einer Teilzeichenkette in einer anderen Zeichenkette lässt sich - schön rekursiv - auch bestimmen. Ist sie nicht vorhanden, dann wird *0* zurückgegeben.



Damit lassen sich Standardoperationen wie das Ersetzen in Zeichenketten leicht realisieren.



Damit wir die Menschheit mit diesen neuen Möglichkeiten beglücken können, exportieren wir die erstellten Blöcke in eine Bibliothek. Dazu wählen wir im Dateimenü *Export blocks …* und wählen dann die zu exportierenden Blöcke aus – natürlich alle! Wir erhalten eine Datei *Stringoperations blocks.xml*, die wir an geeigneter Stelle speichern. Bei Bedarf können wir die Blöcke über das Dateimenü in andere Projekte laden.



13.2 Vigenére-Verschlüsselung

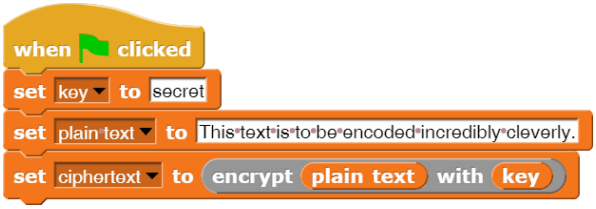
Altersstufe: *Sekundarstufe II* Material: *Vigenere encryption*

Die Vigenére-Verschlüsselung ist eine Erweiterung der Caesar-Verschlüsselung, bei der jedes Zeichen des Klartextes um eine Zahl im Unicode verschoben wird, die sich aus einem Schlüsselzeichen ergibt. Meist ist der Schlüssel kürzer als der zu verschlüsselnde Text, deshalb verlängert man den Schlüssel einfach solange, bis er mindestens so lang wie der Klartext ist.

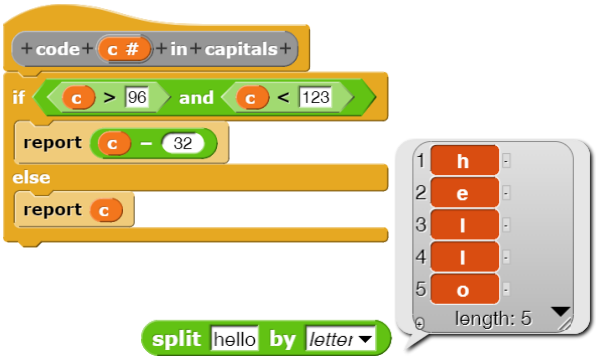
Beispiel:	Klartext:	DASISTEINVOLLGEHEIMERTEXT
	Schlüssel:	NIXDA
	verlängerter Schlüssel:	NIXDANIXDANIXDANIXDANIXDA

Somit wird das erste Zeichen des Klartextes (D) um 14 Zeichen verschoben (N ist das 14. Zeichen), das zweite Zeichen (A) um 9, das dritte (S) um 19 usw. Erhält man Zeichen größer als Z, dann werden die Zeichen zyklisch bei A beginnend verschoben – wie bei der Caesar-Verschlüsselung üblich.

Wir schreiben ein kleines Skript, das den Schlüssel und den Klartext vorgibt und den Geheimtext mithilfe einer Funktion bestimmen lässt. Interessant ist also nur das Verschlüsselungsverfahren.



Da wir mit den Zeichencodes arbeiten, benötigen wir die beiden Blöcke aus der Operators-Palette: *unicode of <…>* und *unicode <…> as letter*.



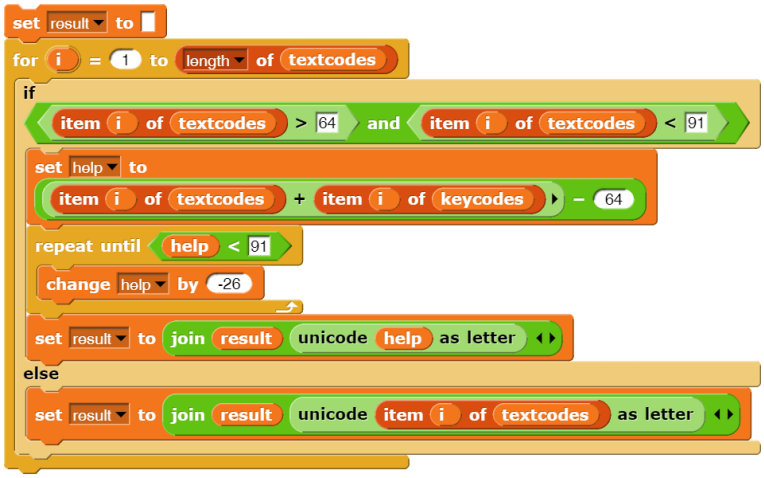
Zuerst einmal wollen wir bei Bedarf Codes aus dem Bereich der Kleinbuchstaben (97 .. 122) in Großbuchstabencodes umwandeln können. Das geschieht, indem bei Bedarf der Wert 32 vom Zeichencode abgezogen wird. Danach erzeugen wir aus dem übergebenen Klartext, einer Zeichenkette, eine Liste von Zeichencodes, die *textcodes* heißen soll. Aus einer Zeichenkette wird eine Liste erzeugt, indem man den *split … by …*-Block anwendet.

Wir übergeben den Code dieser Funktion an den *map <funktionscode> over <liste>* - Block, was man an dem grauen Ring um den Funktionsblock erkennt. D. h. die Funktion wird nicht, wie üblich, zuerst ausgeführt, und dann wird deren Ergebnis übergeben, sondern der Programmcode dieser Funktion wird übergeben, um dann im *map-over*-Block ausgeführt zu werden. In diesem Fall besteht die „gemappte“ Funktion daraus, zuerst den Unicode eines Zeichens zu bestimmen und diesen dann durch die *code in capitals*-Funktion zu schicken. Aus dieser Liste werfen wir noch eventuell ungültige Codes mit einem Wert kleiner 1 heraus. Wir speichern die Code-Listen von Klartext und Schlüssel jeweils in den Variablen *textcodes* und *keycodes*.

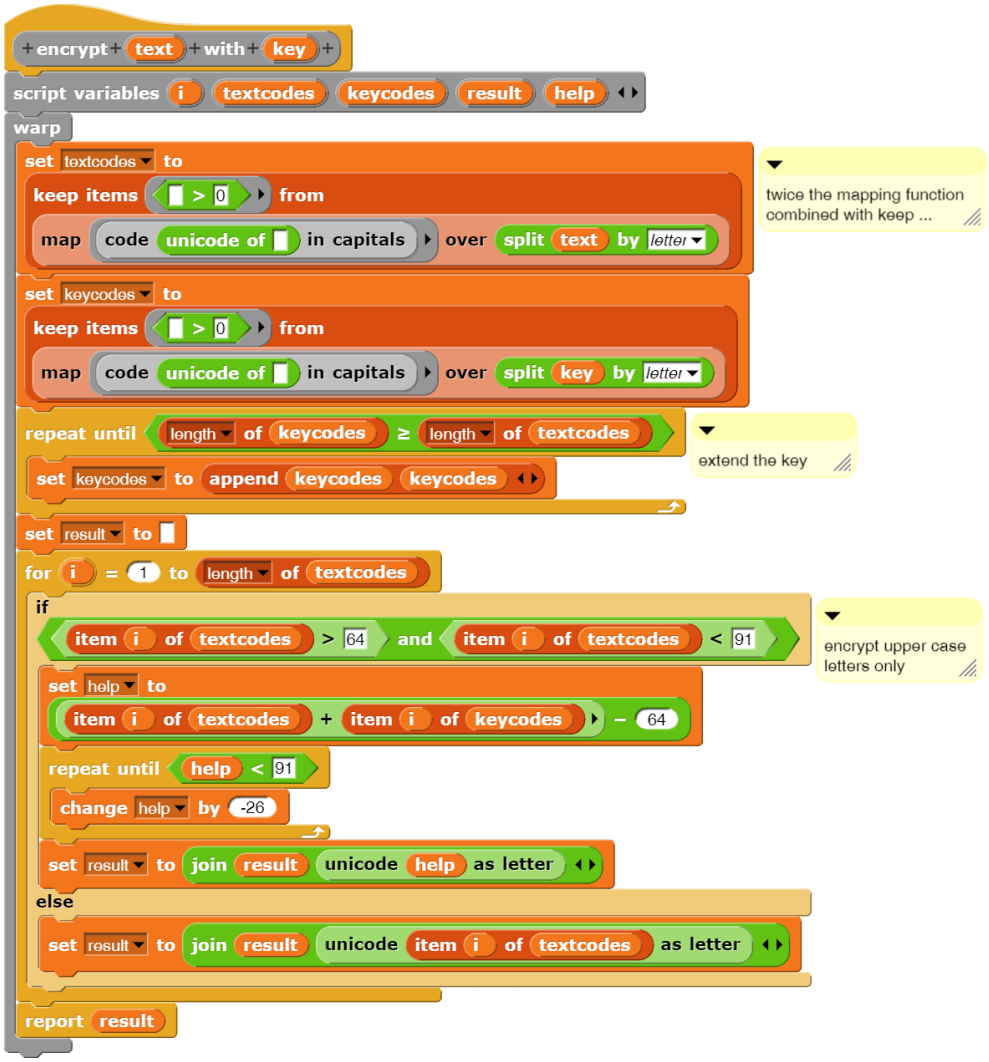
Als Nächstes verlängern wir die *keycodes*-Liste solange um die Codes des Schlüssels, bis die Liste mindestens so lang wie die *textcodes*-Liste ist. Das geschieht hier, indem die *keycodes*-Liste mithilfe des *append*-Blocks jeweils verdoppelt wird.

Jetzt müssen wir nur noch das Vigenére-Verfahren anwenden, in diesem Fall nur auf die Buchstaben. Statt eine Funktion zu „mappen“, verwenden wir diesmal die *For*-Schleife.

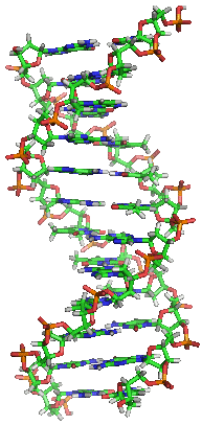
Wir durchlaufen mit ihrer Hilfe alle Zeichen der *textcodes*-Liste und verschlüsseln sie wie angegeben.



Das Verfahren als Ganzes:



13.3 DNA-Sequenzierung[63]

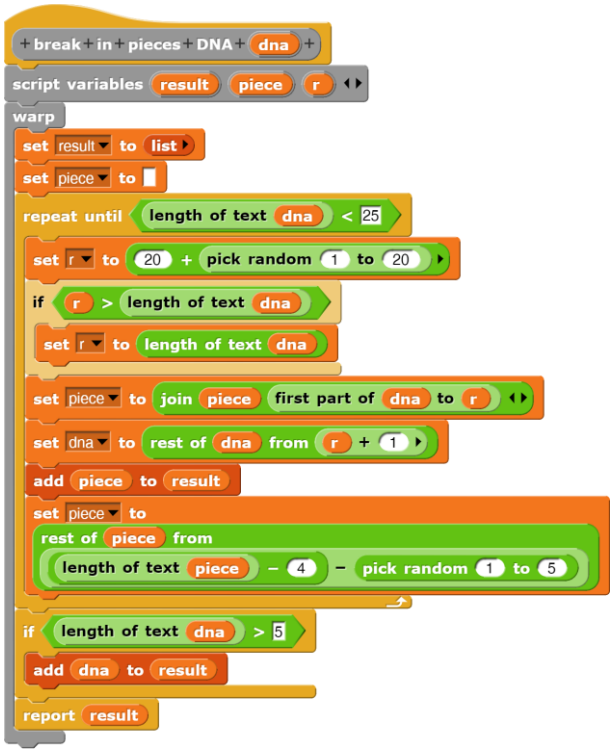


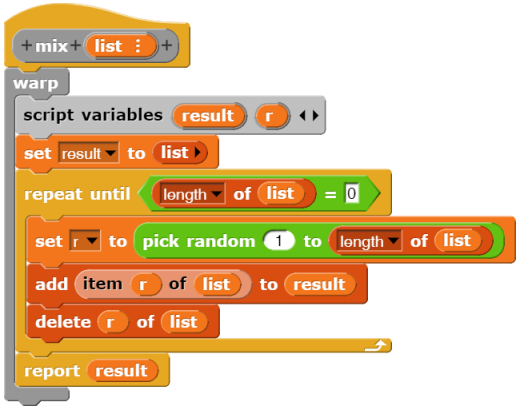
DNA-Helix

Altersstufe: *Sekundarstufe II* Material: *DNA analysis*

In der Bioinformatik werden aus einer Brühe von Biomolekülen, die Teilstücke von DNA-Strängen enthält, Teilsequenzen extrahiert. Aus diesen wird der gesamte DNA-Strang wieder zusammengesetzt. Wir benutzen hier ein stark vereinfachtes Modell, in dem die Teilstücke durch Strings dargestellt werden, die aus den Zeichen A, C, G und T bestehen. Die Bruchstücke „überlappen" sich teilweise, sodass aus Übereinstimmungen an den Kettenenden die ursprüngliche DNA rekonstruiert werden kann.

Zuerst einmal benötigen wir „DNA". Sequenzen findet man im Internet. Da es hier aber nicht auf die Bedeutung der Sequenz ankommt, erzeugen wir sie einfach zufällig. Das Produkt, eine lange Zeichenkette, zerhacken wir, zerlegen sie also in unterschiedlich lange Teilstücke, die sich teilweise überlappen. Wir erledigen diese Aufgabe, indem wir an ein Teilstück vorne ein Stück vom Ende des Vorgängers einfügen. Beim ersten Teilstück ist dieses Stück leer. Wir benutzen die Zeichenketten-Bibliothek, die wir in Kapitel *13.1* angelegt haben.

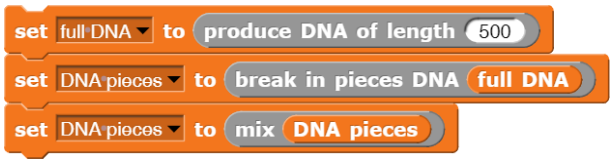




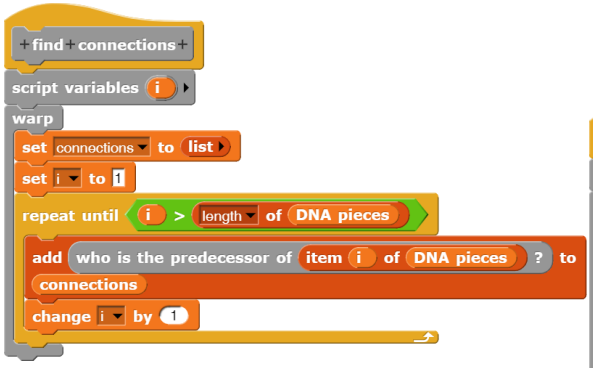
Die Teilstücke befinden sich noch in der richtigen Reihenfolge, sodass die Rekonstruktion kein Problem wäre. Das ändern wir, indem wir die Reihenfolge durcheinanderbringen.

[63] Eine kurze Beschreibung findet sich z. B. unter http://molgen.biologie.uni-mainz.de/Downloads/PDFs/Genomforsch/Modul10B_Skript2015-Hankeln.pdf. Bild aus https://de.wikipedia.org/wiki/Desoxyribonukleinsäure

Mit der folgenden Befehlssequenz erhalten wir dann die gesuchte „Suppe“ aus DNA-Stücken.



Um daraus die ursprüngliche DNA zu rekonstruieren, müssen wir feststellen, welche Bruchstücke einmal aneinanderhingen. Wir erzeugen eine Liste namens *connections*, in die wir jeweils die Vorgänger und die Länge der Überlappung eintragen. Da das erste Teilstück keinen Vorgänger hat, ist dessen Überlappungslänge Null.



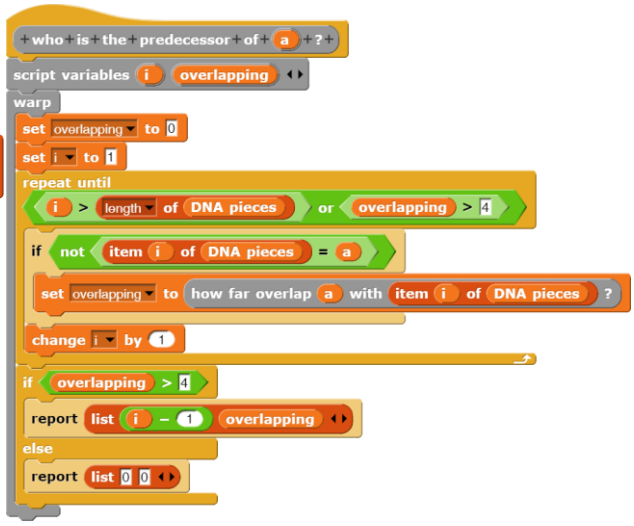
Ein DNA-Stück „hing“ an einem anderen, wenn sich eine genügend lange Überlappung finden lässt. Da Ähnlichkeiten auch zufällig sein können, definieren wir „genügend lang“ als „*5*“. Für eine gegebene Sequenz gibt es für jedes Zeichen vier Möglichkeiten, das richtige Zeichen zu „raten“. Die Wahrscheinlichkeit, das Zeichen zufällig richtig zu erzeugen, ist also *0,25*. Bei fünf Zeichen ist sie dann $0{,}25^5 = 0{,}00098$. Das ist uns genügend „unwahrscheinlich“.

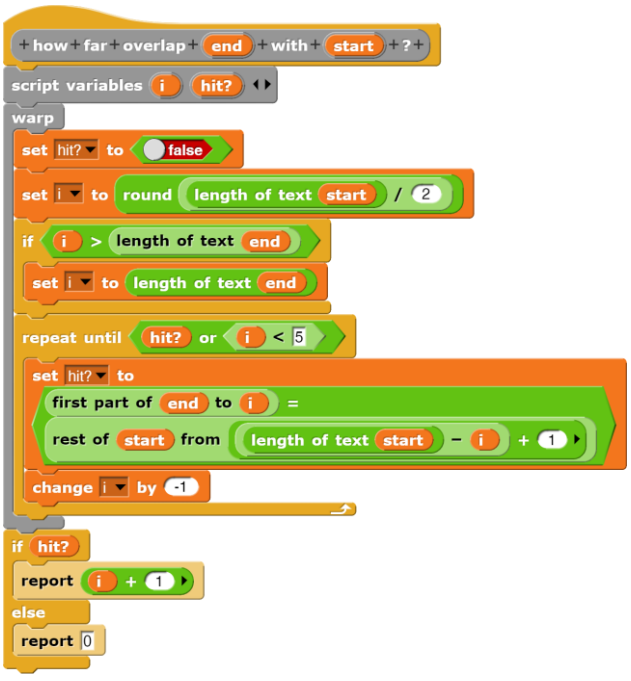
Es bleibt also nur das Problem, festzustellen, ob und wenn, wie weit sich zwei DNA-Sequenzen überlappen. Wir legen sie dazu (gedanklich) ab der Mitte des ersten übereinander und verschieben das zweite dann schrittweise so weit „nach rechts“, bis wir entweder eine Überlappung feststellen oder zu nahe am Ende sind.

Table view

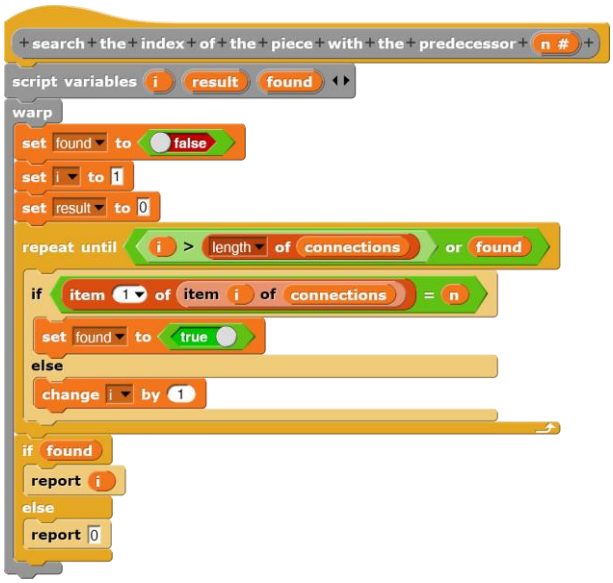
16	A	B
1	12	10
2	4	6
3	2	6
4	5	6
5	7	9
6	16	8
7	14	8
8	15	8
9	0	0
10	0	0
11	8	7
12	3	8
13	9	10
14	13	10
15	6	7
16	1	10

OK

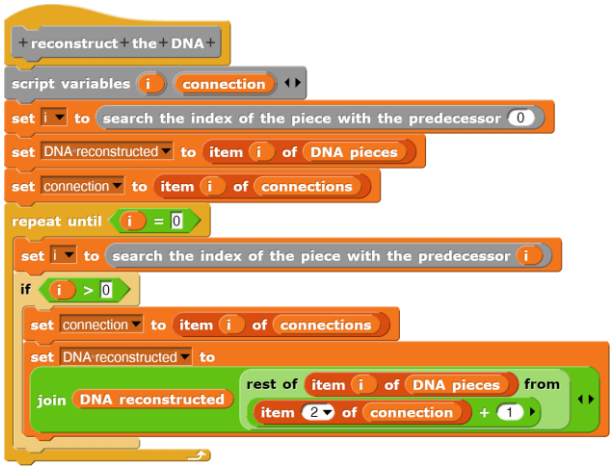




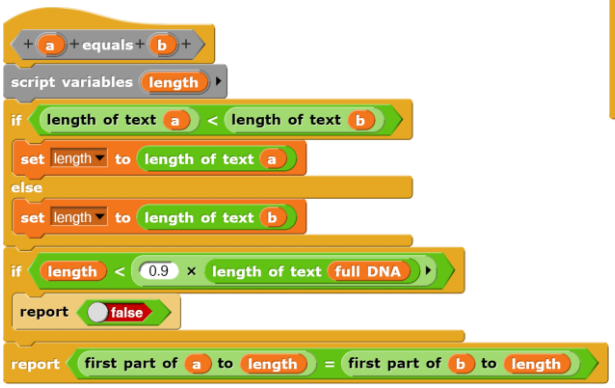
Jetzt sollte man aus der Liste der Verbindungen auch wieder die Original-DNA rekonstruieren können. Wie machen das, indem wir uns – beginnend mit dem Wert 0 des ersten Stücks – durch die Verbindungsliste hangeln. Dafür durchsuchen wir die *connections* nach dem Element, dessen erster Eintrag dem Wert *n* entspricht. Diesen Index geben wir ggf. zurück.



Haben wir ein DNA-Stück gefunden, dann hängen wir es an die bisherigen Fundstücke an und suchen weiter. Der Prozess endet, wenn wir als Fortsetzungsindex eine Null erhalten. Dann sind wir entweder fertig oder es hat sich bei der Suche nach Überlappungen ein Fehler eingeschlichen. Bei der kurzen Überlappungslänge geschieht das schon einmal. 😉



Zuletzt überprüfen wir, ob es mit der DNA-Rekonstruktion geklappt hat. Die rekonstruierte DNA sollte schon annähernd so lang sein wir das Original – und natürlich sollte sie in diesem Bereich auch mit der Vorlage übereinstimmen.

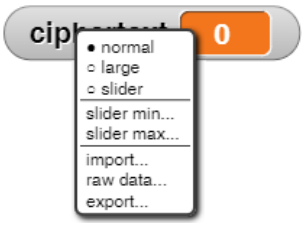


Tut sie auch – meistens.

13.4 Textdateien, Server und Häufigkeitsanalyse

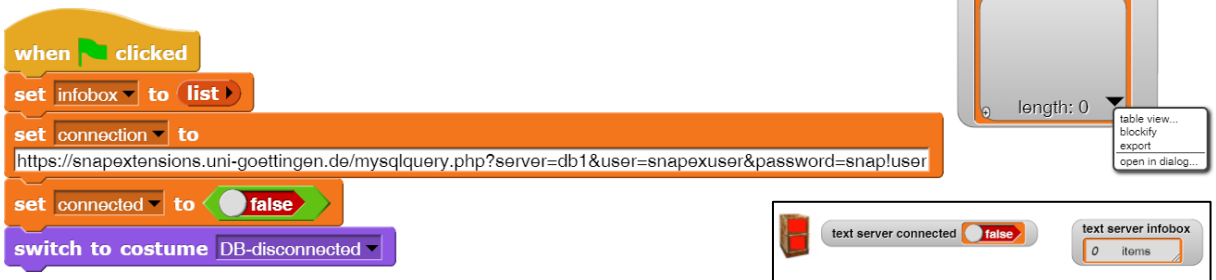
Altersstufe: *Sekundarstufe II* Material: *Textfiles server and frequency analysis*

Aus undurchsichtigen Quellen haben wir die Information bekommen, dass sich ein unglaublich geheimer Text in der Datei *ciphertext.txt* auf unserem Rechner befindet. Wir erfahren sogar, in welchem Verzeichnis er sich befindet. Um den Text von *Snap!* aus bearbeiten zu können, erstellen wir eine Variable *ciphertext* und lassen sie im Arbeitsbereich anzeigen. Als Inhalt zeigt sie die Null. Wir wählen aus dem Kontextmenü der angezeigten Variablen den Punkt *import…*, navigieren zum genannten Verzeichnis und wählen den geheimen Text aus. Er erscheint in der Variablen.



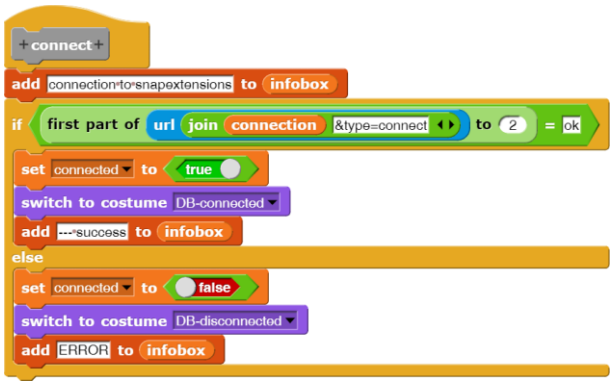
Sicherheitshalber wollen wir den Text an anderer Stelle sofort wieder speichern. Wir wählen aus demselben Kontextmenü den Punkt *export…* und erhalten unten-links im Fenster ähnlich wie beim Speichern eines Projekts die Datei *ciphertext.txt*. Diese finden wir im Download-Verzeichnis unseres Rechners. Das geschilderte Verfahren ist einfach, kann allerdings nicht vom Programm gesteuert werden, sondern wird „per Hand“ ausgeführt.

Textdateien sind ein einfaches, aber zuverlässiges Werkzeug, um Daten zwischen verschiedenen Rechnern auszutauschen. Damit das geht, benötigen wir einen *http-Server* (der ggf. auch der gleiche Rechner sein darf), auf dem ein Skript läuft, das die gewünschte Funktionalität besitzt – hier: das Laden und Speichern von Textdateien. In diesem Fall wollen wir den Server *snapextensions.uni-goettingen.de* wählen, auf dem sich das Skript *handleTextfile.php* befindet. Wir zeichnen zwei Kostüme für ein *Textserver*-Sprite, die anzeigen, ob wir mit dem Server verbunden sind – oder nicht. Der Datenaustausch mit dem Server soll in einer Variablen *infobox* protokolliert werden. Durch Anklicken der grünen Flagge sollen unsere Variablen initialisiert werden, wobei die namens *connection* einen ziemlich kryptischen Wert erhält.

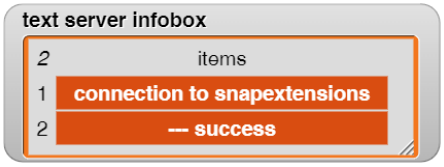


Dieser besteht aus der Adresse des Servers, einem Anmeldeskript und einigen Variablen – PHP eben. Unsere Infobox stellen wir mithilfe des Kontextmenüs auf „*table view*“ um, was etwas besser aussieht. Das Ausgabefenster stellt sich danach so dar:

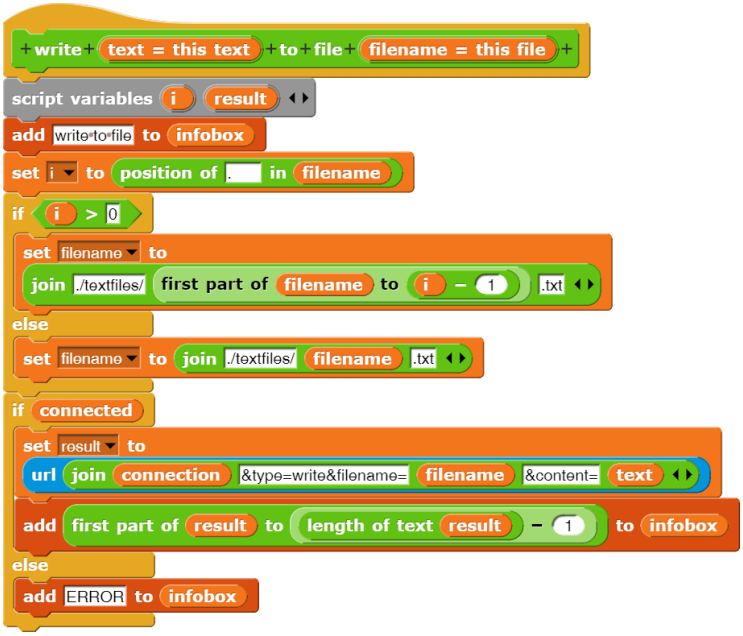
Wir benötigen eine Verbindung zum Server. Das geschieht mithilfe des *url*-Blocks, dem wir die erforderlichen Daten übergeben. Den Erfolg oder Misserfolg protokollieren wir in der Infobox.



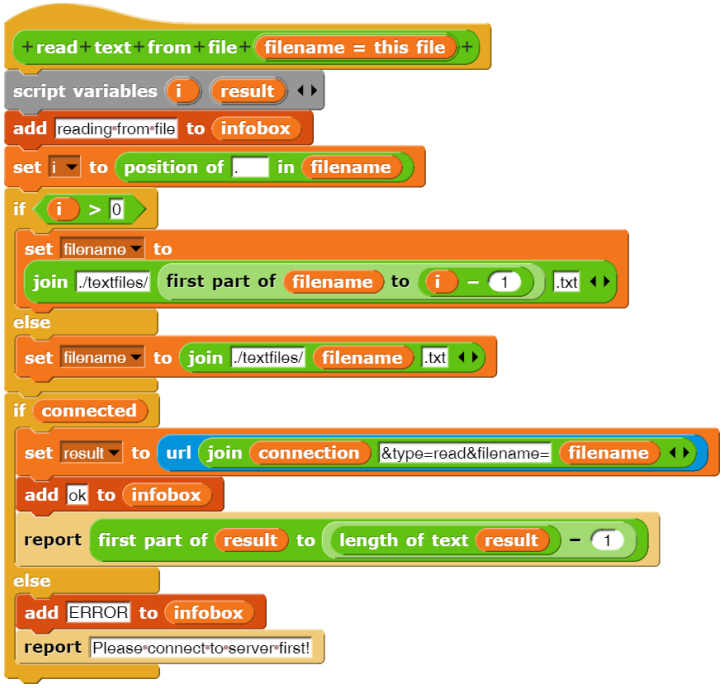
Nach Ausführung dieses Blocks steht zwar die Verbindung zum Server, aber der Text in unserer Infobox ist nur teilweise zu sehen. Wir klicken deshalb mit der linken Maustaste auf die Spaltenüberschrift *items* und ziehen die Spalte in die Breite, bis aller Text lesbar ist.



Wir wollen Daten in eine Datei auf dem Server schreiben. Den zu schreibenden Text und den Dateinamen geben wir als Parameter an. Zuerst hängen wir bei Bedarf die Endung *„.txt“* an den Dateinamen an und sorgen dafür, dass die Datei im Unterverzeichnis *textfiles* auf dem Server gespeichert wird. Dann übermittelt der *url*-Block die erforderlichen Daten.

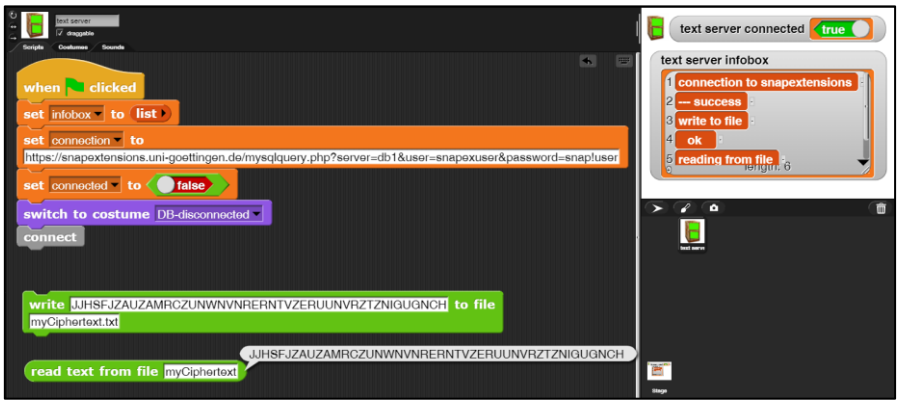


Das Lesen aus einer Datei erfolgt entsprechend.



Das Textserver-Sprite exportieren wir in eine XML-Datei und können so dessen Funktionalität auch in anderen Projekten nutzen.

Nach einem Verbindungsaufbau, einem Schreib- und einem Lesevorgang sieht unser Arbeitsbereich etwa so aus:



Es hilft nichts, wir müssen den Geheimtext jetzt entschlüsseln. Dazu führen wir eine Häufigkeitsanalyse durch – d. h. wir zählen, wie oft die einzelnen Buchstaben in einem Text vorkommen.

Da im Deutschen das *E* der häufigste Buchstabe ist und es gemein wäre, wenn der Text in einer anderen Sprache verfasst worden wäre, speichern wir die Liste der Häufigkeiten in einer Variablen *frequencies*.

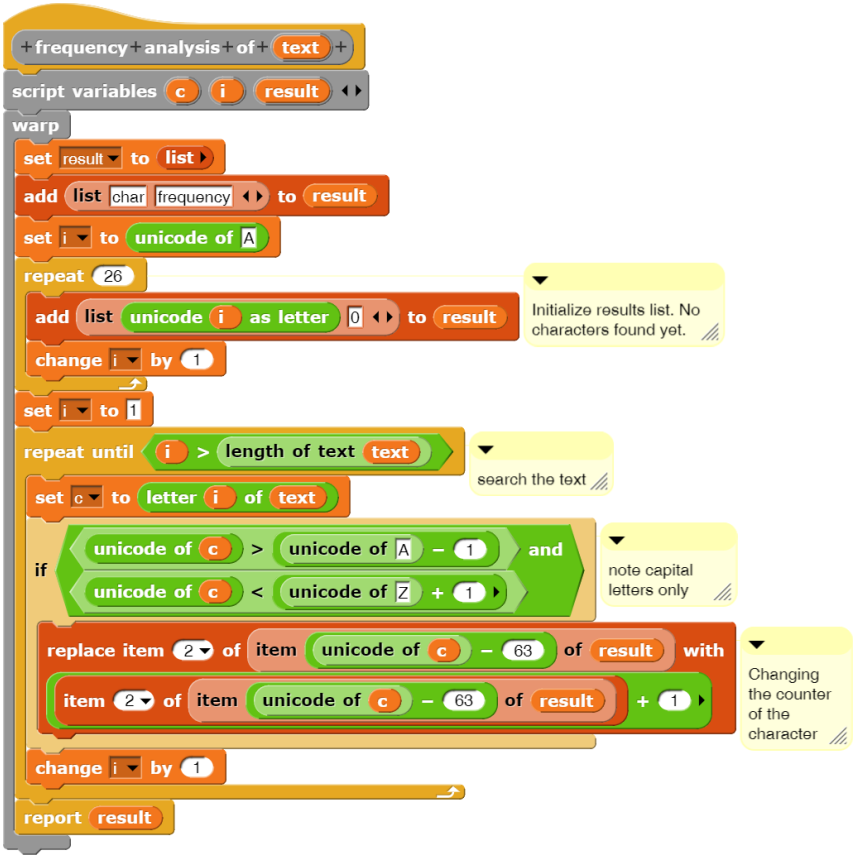


Table view

27	A	B
1	char	frequency
2	A	94
3	B	159
4	C	46
5	D	40
6	E	433
7	F	248
8	G	570
9	H	4
10	I	380
11	J	51
12	K	182
13	L	290
14	M	283
15	N	113
16	O	364
17	P	2
18	Q	269
19	R	214
20	S	151
21	T	1034
22	U	62
23	V	131
24	W	2
25	X	303
26	Y	12
27	Z	79

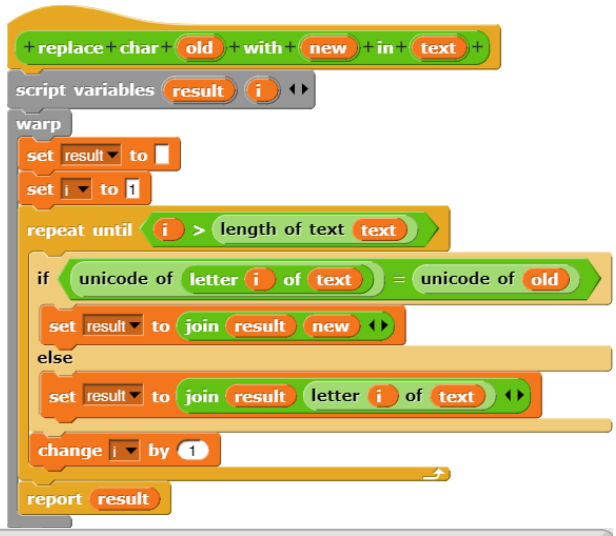
OK

Danach versuchen wir, das große *T* im Geheimtext durch ein kleines *e* zu ersetzen– denn *T* kommt ja am häufigsten vor. Für so viele Ersetzungen ist unser *replace*-Block nun wirklich nicht gedacht, deshalb schreiben wir schnell einen neuen. In diesem unterscheiden wir bei Textvergleichen Groß- und Kleinschreibung und benutzen deshalb die *Unicodes* der Zeichen.

Weil das Ergebnis nicht allzu beeindruckend ist, benötigen wir mehr Ersetzungen. Wir vermuten hinter dem *G* ein *n* und führen auch diese Ersetzung durch.

Den Geheimtext können wir uns ganz gut ansehen, wenn wir ihn mit 

in Zeilen zerlegen.



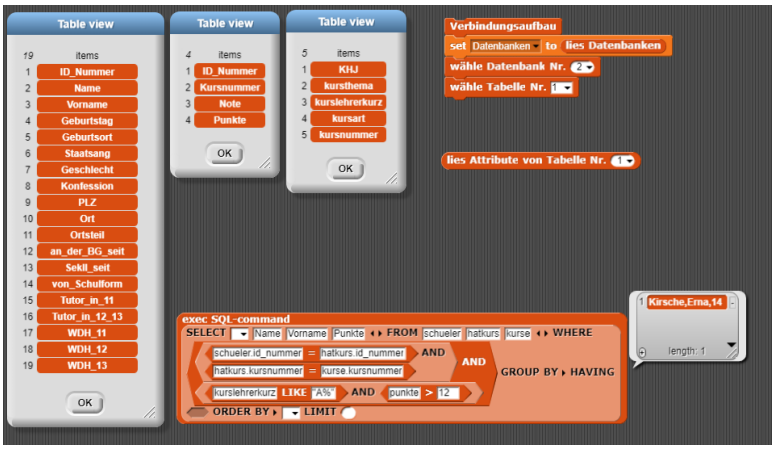

```
1  QEeLeO ZFBMI /YSMXnn ASKZRXnR USn RSeLMe
2  eEn NeORUXRXNFnQen-ReIDOXeBM VEL eEneV VeMOZXBMen eFOSDXeEIBMen REDZeKILFeOVeO
3  MeOO ILXXLIVEnEILeO USn RSeLMe, IEe MXNen KXnRYXeMOERe NeORILeEReOEIBMe eOZXMOFnRen.
4  YX, EBM MXNe QEe ZFOCX, Qen RSLLMXOQ NeILEeRen! QEeIeO eOMXNenen, FnUeORKeEBMKEBMen nXLFOIJenen AeOQen E
5  RXN el nSBM XnQeOe IBMAeEJeO REDZeK, QEe EV UeOKXFZ EMOeO QOeE OeEIen En QXI KXnQ QeO eEQRenSIIen USn EMner
6  EBM AXO En eEnEReO ISORe AeRen Qel neNeKI, QSBM JSR EBM XFI QeO ReILXKL Qel SNeOn MEVVeKI eEnERe RFLe USONeC
7  AXO EMnen IS AeEL SNen nEeVXKI IBMAEnQKER, EBM VeEne, KELLen IEe nEe Xn MSeMenXnRIL?
8  Aenn YX, AEe CSnnLen IEe QEeIe FeNeOAEnQen? EBM eOILEeR RXnJ XKKeEn Qen MSeBMILen REDZeK Qel (ILOXIINFOReO) V
9  XKKeV XnIBMeEn nXBM EnLeOeIIEeOLen IEe IEBM NeISnQeOI ZFeO Qen RSLLMXOQ. AXOFV?
10 QeO RSLLMXOQ EIL JAXO nEBML QXI MSeBMILe ReNEOR QeO IBMAeEJ, FnQ En IXUSHeO FeNeOLOEZZL EMn QeO VSnLNKXr
11 XFBM En ELXKEen RENL el MeOOKEBMe NeORe, JFVXK QEe UFKCXne. XFZ QeO DOSZEKCXOLe IEnQ UeIFU FnQ XeLnX FnIBM
12 QEe CFnQe eEneO ISeNen XFINOeBMenQen KXUX, QEe ZFeO neXDeK FnIEBMLNXO nXBM SLLXYXnS MEnFnLeOZKEeIIL, OeEJ
13 ... FnQ Qen RnDZeK JF NeJAEnRen!?
14 XV ZFIIe Qel ILeEKen MXnReI eVDZEnRen FnI JAeE ZFeMOeO, eEn XeKLeOeO FnQ eEn YFenReOeO, NeEQel LFeBMLERe KeFLe.
```

Wir finden z. B. Worte wie *eEn* in Zeile 2. Wir halten deshalb das *E* für ein *i*.

Das war eine gute Idee! Suchen wir weiter und probieren Ersetzungen aus, dann finden wir irgendwann auch das Geheimnis! Man muss nur durchhalten – es sind ja nur noch 23 Buchstaben!

13.5 SQL-Datenbanken

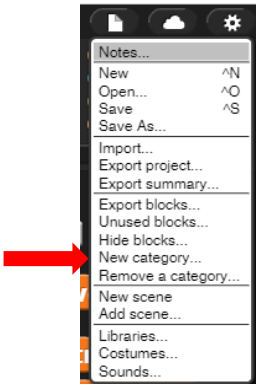
Altersstufe: *Sekundarstufe I/II*
Materialien: *SQL, SQL example*



Eine wichtige Anwendung von Informatiksystemen ist der Zugriff auf externe Datenquellen. Dafür steht einerseits das Internet zur Verfügung, andererseits ist die Benutzung von SQL-Datenbanken üblich. Da die Nutzung dieser Art von Anwendung in vielen Computersprachen etwas kompliziert ist, wird sie oft von der Algorithmik getrennt behandelt. Dadurch wird dieses Teilgebiet der Informatik ziemlich trocken: man erstellt ER-Diagramme auf dem Papier oder befragt Datenbanken mit speziellen Client-Anwendungen, z. B. mit *PHPmyAdmin*, nutzt die Ergebnisse aber nicht weiter aus. Mithilfe von *Snap!* geht das auch anders!

Wir benötigen wieder einen Server, der entweder auf einem anderen Rechner oder auch auf dem eigenen läuft, und auf dem sich – in diesem Fall – neben einem http-Server und einem SQL-Server[64] ein PHP-Skript namens *mysqlquery.php* befindet, dem wir mithilfe der Parameter *type*, *query*, *command*, … die für eine SQL-Anfrage an den SQL-Server benötigten Daten übermitteln. Das Ergebnis der Anfrage ist dann entweder eine Fehlermeldung oder eine Tabelle mit Ergebnissen. Diese wird vom Skript bei Bedarf etwas aufbereitet, sodass sie von *Snap!* als Liste angezeigt werden kann. Der Quelltext dieses Skripts findet sich z. B. auf https://snapextensions.uni-goettingen.de.

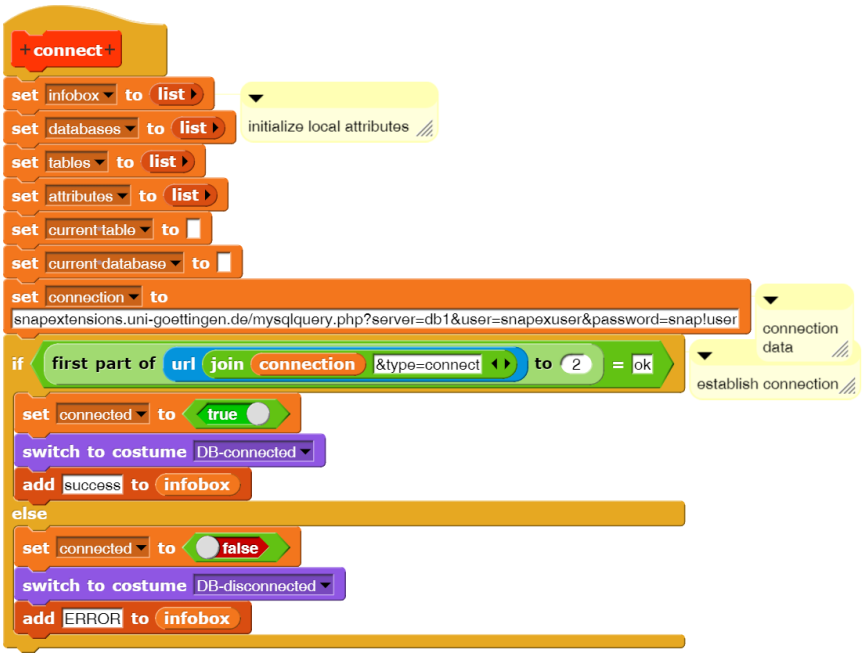
Wir erzeugen ähnlich wie im letzten Abschnitt ein Sprite namens *SQLserver*, das durch sein Kostüm anzeigt, ob eine Verbindung zur Datenbank besteht. Einige Attribute wie *connection*, *connected*, *current table* usw. speichern den aktuellen Zustand, und in einer Variablen *infobox* werden die Abläufe protokolliert. Dieses Sprite wird als *SQLserver* gespeichert und kann bei Bedarf geladen werden.



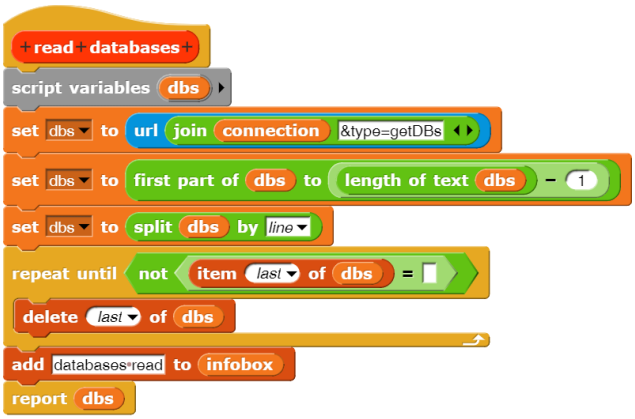
Die für SQL-Abfragen nötigen neuen Blöcke werden global vereinbart, damit sie leicht für Abfragen außerhalb des Server-Sprites zugänglich sind. Sie werden in der Datei *SQL blocks.xml* gespeichert und müssen zusätzlich geladen werden. Da es sich um eine ganz andere Kategorie von Blöcken handelt als die in den Standardpaletten vorhandenen, spendieren wir dem System eine neue Palette namens SQL, in der wir die SQL-Blöcke unterbringen.

[64] Im Projekt „Im Supermarkt“ wird auch ein SQLite-Server benutzt.

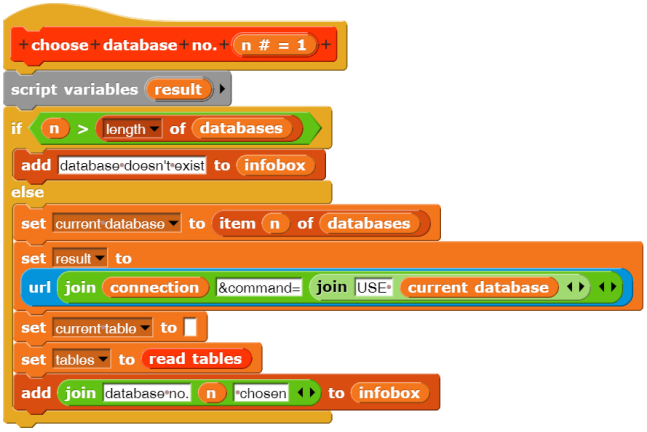
Zuerst einmal benötigen wir einen Zugang zum externen SQL-Server. Dafür richten wir einen Block *connect* ein. Dort werden die lokalen Attribute initialisiert und die Verbindungsdaten werden in der Variablen *connection* gespeichert, damit sie nicht jedes Mal neu eingegeben werden müssen. Danach wird die Verbindung aufgebaut und der Erfolg bzw. Misserfolg in der Variablen *connected* notiert.



Mithilfe des Reporter-Blocks *read databases* wird der SQL-Server nach den vorhandenen Datenbanken befragt. Diese werden als Liste zurückgeben. Für die eigentliche Abfrage ist nur noch als „type“ der Wert „getDBs“ an die Verbindungsdaten anzuhängen.

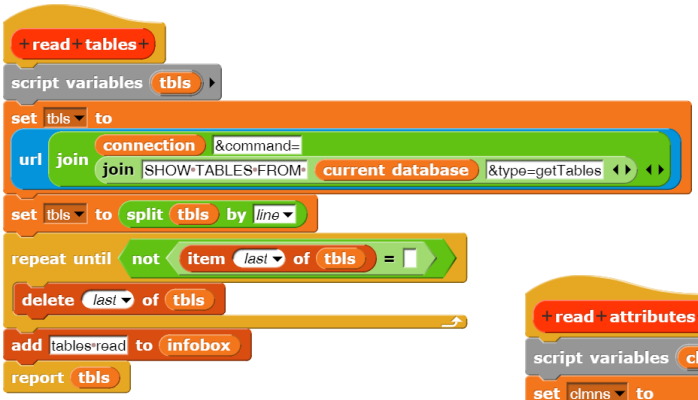


Der Verbindungsaufbau und die Auswahl einer Datenbank kann als Blocksequenz gespeichert werden. Der letzte Block wählt die angegebene Datenbank aus. Interessant dabei ist der kleine Pfeil neben dem Parameter. Klickt man den an, dann erscheint eine Auswahlliste mit den möglichen Werten.

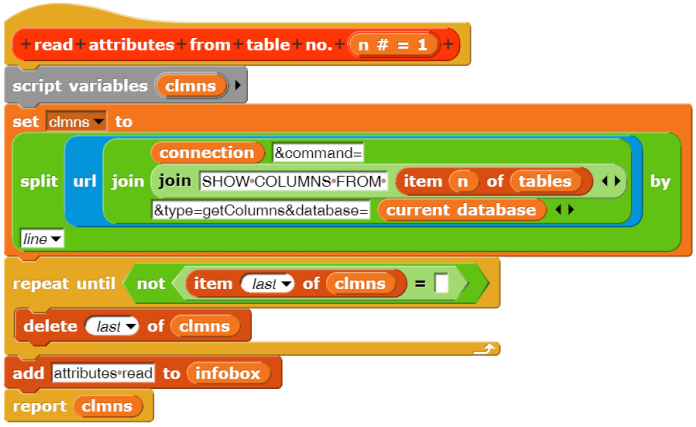


Eine Auswahlliste kann im Blockeditor erzeugt werden, wenn man einen Rechtsklick im dunklen Bereich ausführt. Man erhält dann ein kleines Kontextmenü mit dem Punkt *options…* Im aufklappenden Fenster *Input Slot Options* werden die möglichen Eingabeoptionen eingegeben.

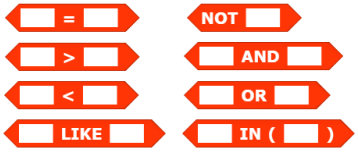
Auf ganz ähnliche Weise wird für die gewählte Datenbank ermittelt, welche Tabellen sie enthält und aus welchen Attributen die Tabellen aufgebaut sind.



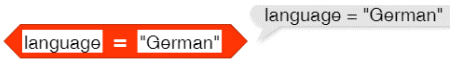
Mithilfe der neuen Blocks können wir also erfahren, welche Tabellen vorliegen und welche Attribute sie enthalten. Im Kontextmenü der erhaltenen Liste lässt sich das Ergebnis dauerhaft über die Option „open in dialog“ anzeigen. Auf diese Weise kann man die für Anfragen benötigten Werte übersichtlich am Bildschirm zusammenstellen.



Wir haben jetzt die Voraussetzungen geschaffen, um Anfragen an die Datenbank zu stellen. Dafür benötigen wir noch SQL-Aggregatfunktionen und -Operatoren. Aus diesen lassen sich mithilfe der Daten aus den „Table views“ und zwei Arten von SELECT-Blöcken SQL-Anfragen interaktiv zusammenstellen.

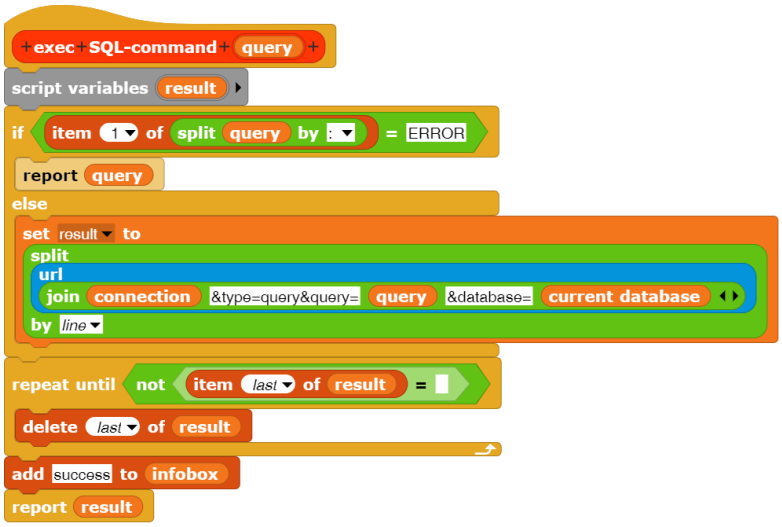


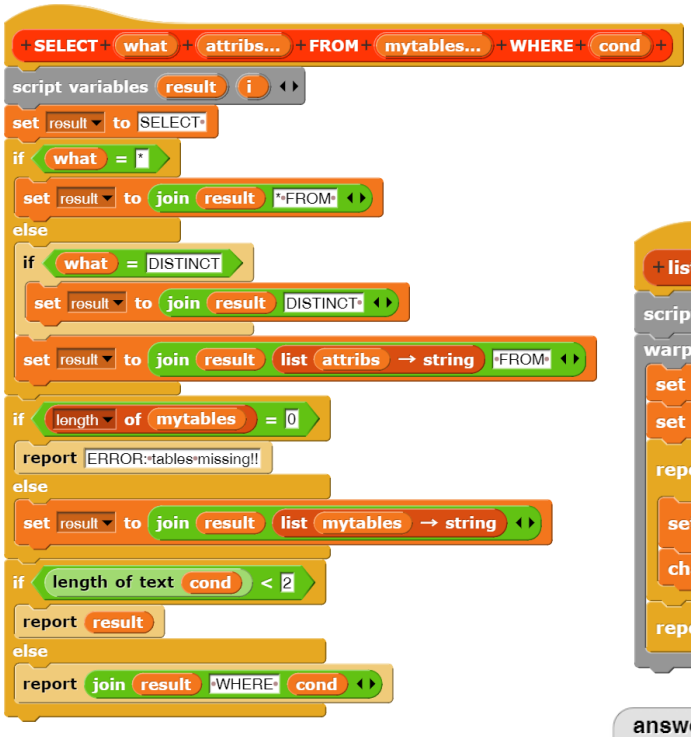
Zu beachten ist, dass von den Blöcken nur die Texte der Anfragen erzeugt werden! Die Anfragen werden (noch) nicht ausgeführt.



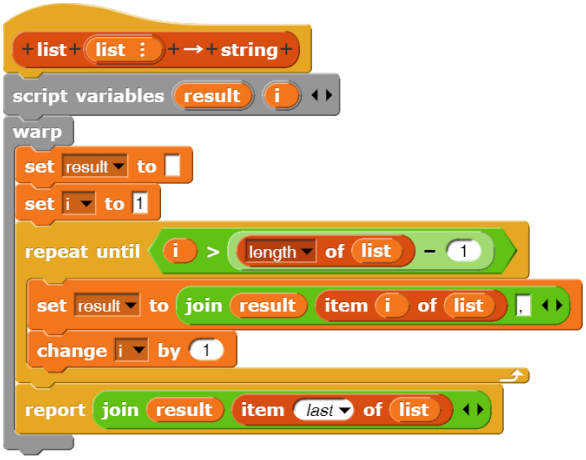
Mit diesen Blöcken lassen sich jetzt SELECT-Anfragen erstellen – und kontrollieren.

Zur Ausführung solcher Anfragen haben wir einen – letzten – Block zur Verfügung. Diesem wird ein SQL- Befehl entweder als Text oder als Ergebnis eines SELECT-Blocks übergeben. In der erhaltenen Antwortliste werden eventuelle leere Einträge gelöscht.



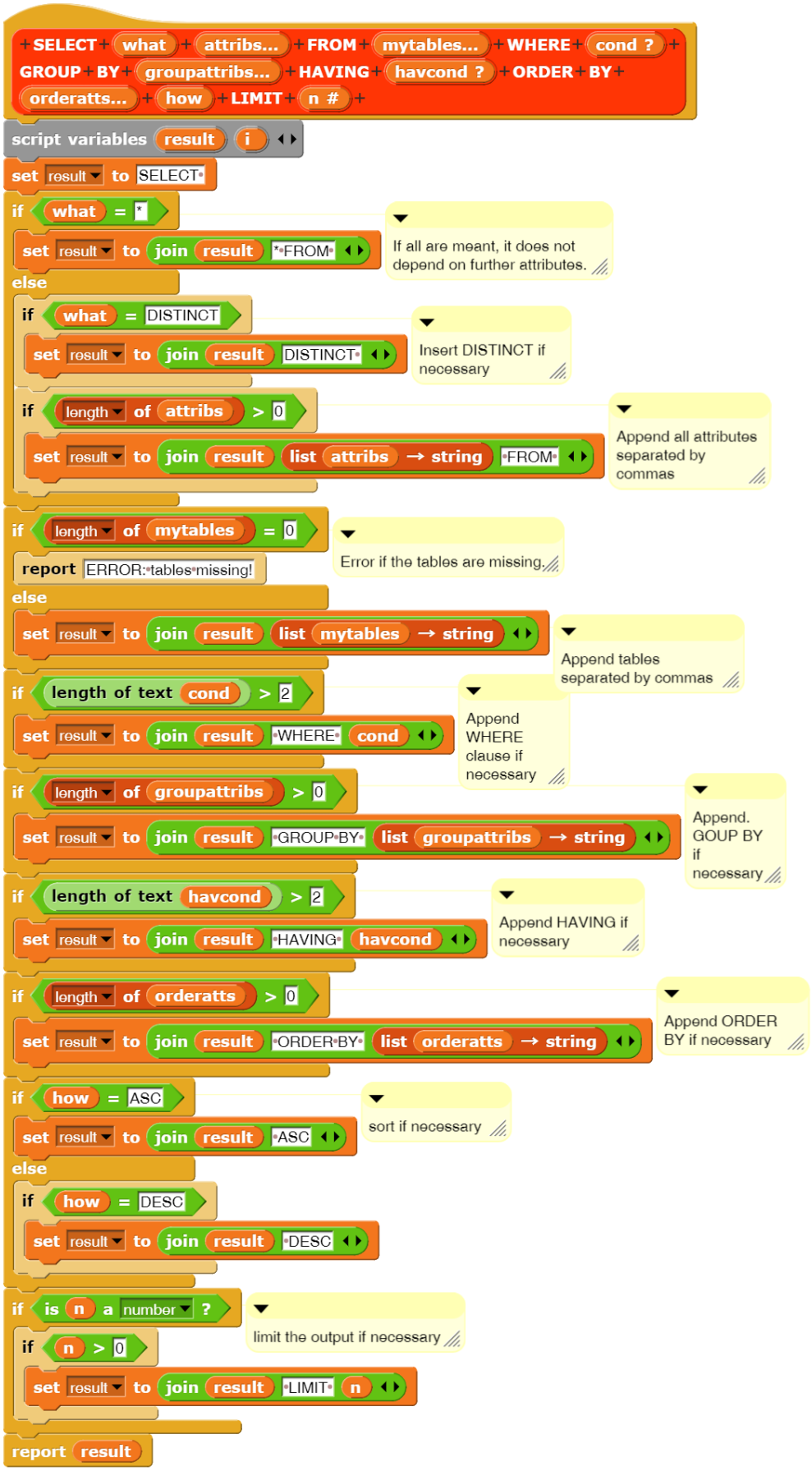


Der einfache SELECT-Block baut aus den Parametern eine SQL-Anfrage zusammen. Er nutzt dafür einen Reporter *List → string*.

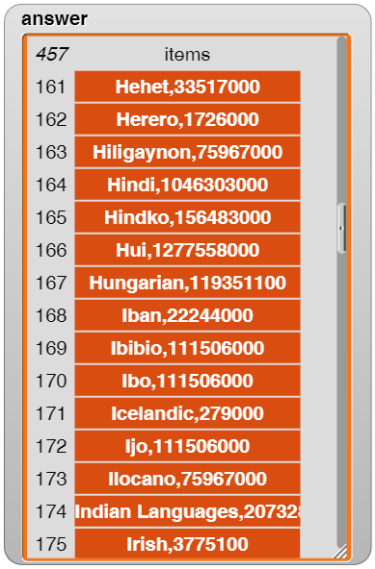


Damit können wir einen ersten Versuch starten:

Beim vollen SELECT-Block ist das auch nicht komplizierter – nur länger.



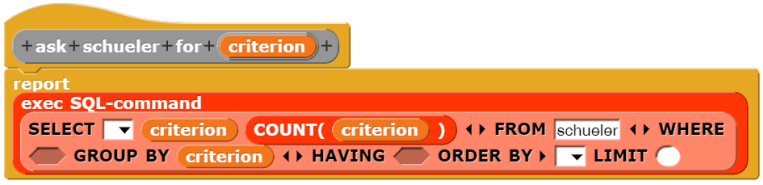
Damit können wir jetzt arbeiten: Wieviel Menschen sprechen welche Sprache?



Erstaunlich!

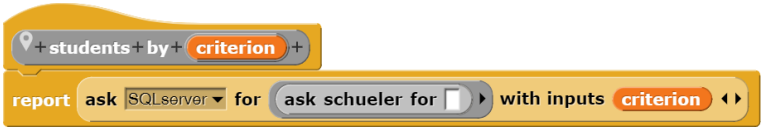
Die entstandene SQL-Bibliothek ist dafür gedacht, SQL-Befehle interaktiv zu erproben und sie dann – bei Erfolg – in neue Blöcke einzubinden, die die Nutzung der Datenbank ohne SQL-Kenntnisse ermöglichen. Wir verdeutlichen das anhand einer einfachen Anfrage.

Für ein neues Projekt importieren wir zuerst die *SQL blocks*-Bibliothek, danach das *SQL-Server*-Sprite. Zusätzlich erzeugen wir ein *SQL-Nutzer*-Sprite. Dieses bittet dann den SQL-Server um einen Verbindungsaufbau.



Danach können Abfrageblöcke erstellt werden, die z. B. die für eine Schulstatistik wichtigen Daten ermitteln.

Die können dann für spezielle Zwecke eingesetzt werden.



13.6 Aufgaben

1. Eine einfache Form des **Blockchiffrierens** besteht darin, den zu verschlüsselnden Text in eine Tabelle mit mehreren Spalten von links nach rechts und von oben nach unten einzufügen. Ist die letzte Zeile nicht gefüllt, dann werden irgendwelche Zeichen eingesetzt. Der verschlüsselte Text ergibt sich, indem man die Tabelle von oben nach unten und von links nach rechts ausliest.

   Beispiel:

   ```
   DIESE      →      DRIHIHIHITSECEEETIHISXUMGMETNLEX
   RTEXT
   ISTUN
   HEIML
   ICHGE
   HEIMX
   ```

   Worin besteht der Schlüssel? Realisieren Sie das Verfahren.

3. **Genetische Algorithmen** simulieren das evolutionäre Vorgehen der Natur, indem sie zufallsgesteuert immer wieder neue Kandidaten zur Lösung eines Problems erzeugen. In diesem Fall sind **Palindrome** gesucht, also Worte, die vorwärts und rückwärts gelesen gleich sind. Das Verfahren besteht aus einer Initialisierung, in der eine zufällige Anfangspopulation erzeugt wird. In diesem Fall eine Menge von Zufallsworten. Danach wird eine Schleife immer wieder durchlaufen, in der aufgrund einer Fitnessfunktion Kandidaten für eine Rekombination von Individuen ausgewählt werden. Aus zwei Kandidaten wird (mindestens) ein neuer erzeugt. Danach erfolgen zufällige Änderungen (Mutationen). In der entstandenen neuen Generation werden aufgrund der Fitnessfunktion die „besten“ Kandidaten für den nächsten Durchgang ausgewählt (Selektion).

4. Die Bestimmung der **Levenshtein-Distanz** zwischen zwei Zeichenketten dient dazu, den „Verwandtschaftsgrad“ der Zeichenketten festzustellen. Typischerweise handelt es sich bei diesen um DNA-Stränge aus den Zeichen A, C, G und T.

   a: Informieren Sie sich über das Verfahren.

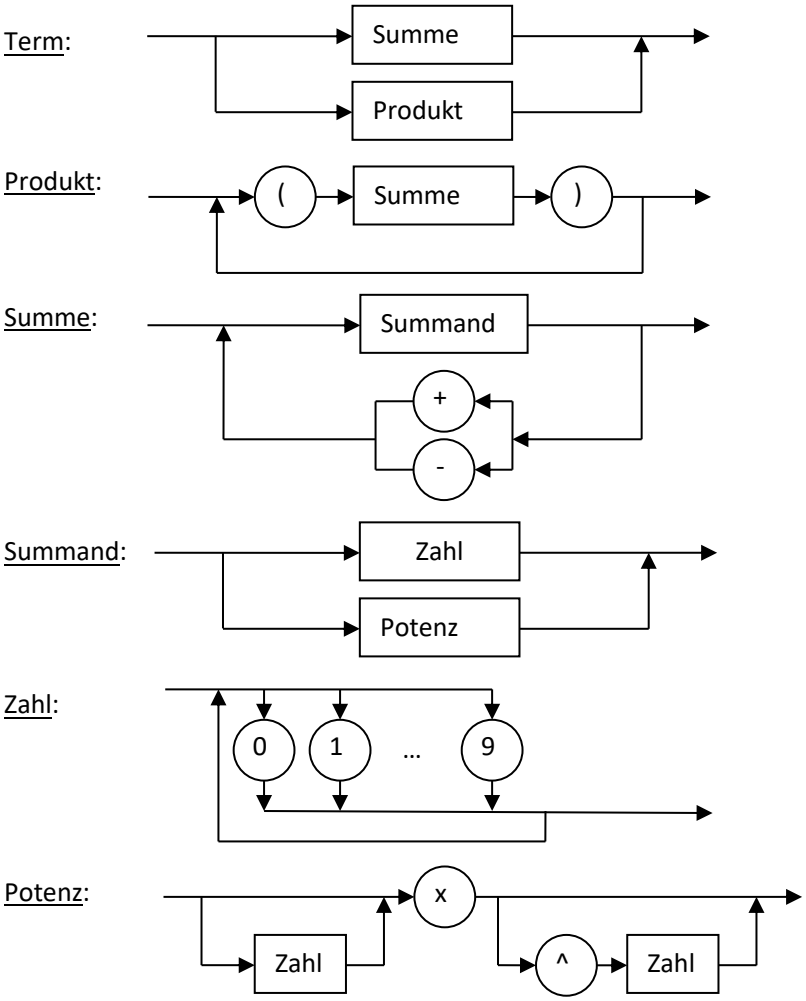
   b: Realisieren Sie das Verfahren.

14 Computeralgebra: funktional programmieren

Altersstufe: *Sekundarstufe II* Material: *Computer algebra*

14.1 Funktionsterme

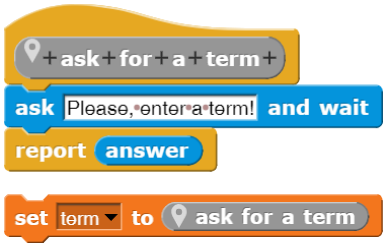
Wir wollen ein kleines „*Computeralgebrasystem*“ (CAS) entwickeln, das einerseits die *Top-down-Methode* illustriert und andererseits zeigt, wie man mit *Snap!* funktional programmiert. Dazu müssen wir z. B. über Syntaxdiagramme definieren, was wir unter Funktionstermen verstehen wollen.



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

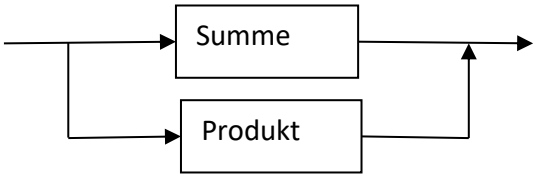
14.2 Funktionsterme parsen

Zur Arbeit mit Funktionstermen brauchen wir natürlich jemanden, der etwas davon versteht. Wir zeichnen deshalb *Gundolf de Jong*, einen begabten jungen Mathematiker, und machen den danach schlau. Zuerst einmal muss Gundolf Funktionsterme einlesen können. Dafür bittet er den Benutzer um eine entsprechende Eingabe mithilfe des Blocks *ask <question> and wait* aus der *Sensing*-Palette. Das Ganze verlagern wir in eine Methode Gundolfs, die wir als Funktion definieren. Wir wählen also die ovale Blockform im Blockeditor aus. Haben wir eine Variable, z. B. namens *term*, vereinbart, dann können wir dieser das Ergebnis der Eingabe zuweisen.

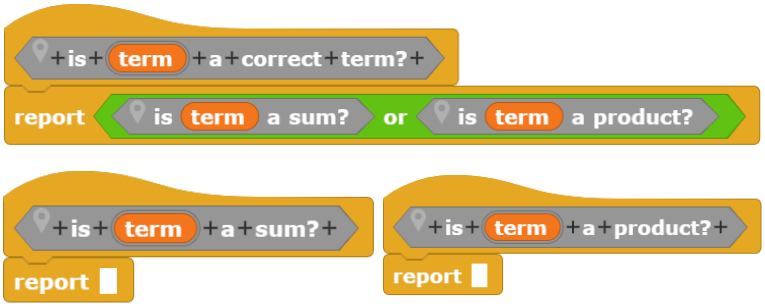


Als nächstes überprüfen wir, ob die Eingabe korrekt ist. Die entsprechenden Methoden verlagern wir in ein Sprite namens *Parser*. In diesem wollen wir einerseits funktional programmieren, andererseits das Problem auf dem Top-Down-Weg lösen.

Wir erzeugen den globalen Block (*for all sprites*) *is <term> a correct term?* als Prädikat, das dementsprechend nur die Resultate *true* oder *false* liefern kann. Danach haben wir zwar einen schönen Titel, aber leider noch keinen Inhalt. Trotzdem können wir den Block schon in Skripten verwenden – genauso wie weitere Blöcke. Das ermöglicht einerseits rekursive Operationen, andererseits eignet es sich zur Top-Down-Entwicklung. Da laut der Syntaxdiagramme korrekte Terme entweder Summen oder Produkte sind, verschieben wir das Problem dorthin, indem wir zwei entsprechende Prädikate erzeugen – immer noch leere –, und zwar lokal (*for this sprite only*), weil der Rest des Problems externe Beobachter nichts angeht.



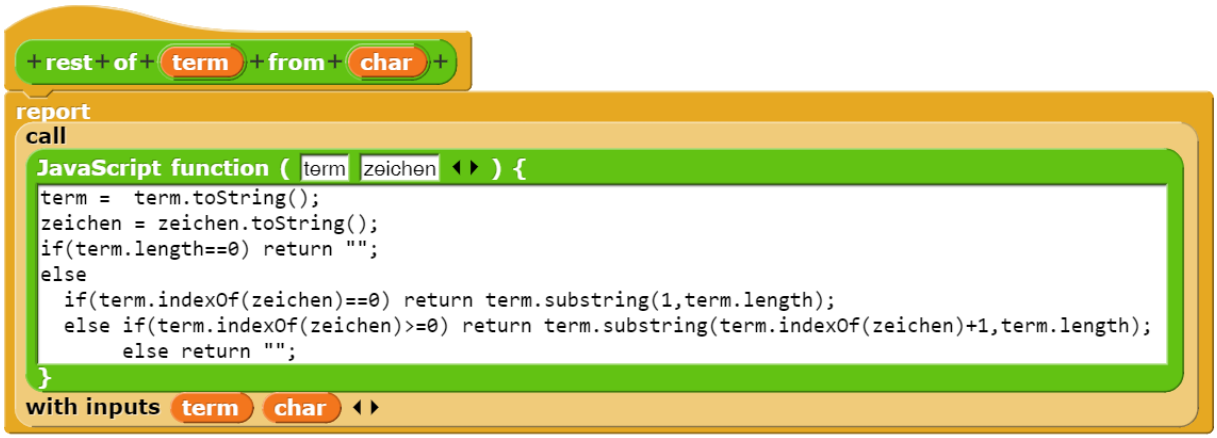
Snap! wertet logische Ausdrücke inzwischen „lazy“ aus: der zweite Ausdruck wird nur ausgewertet, wenn der erste nicht schon das Ergebnis bestimmt. Wir können das Prädikat *is <term> a correct term?* deshalb vollständig mit leeren Blockrümpfen angeben.

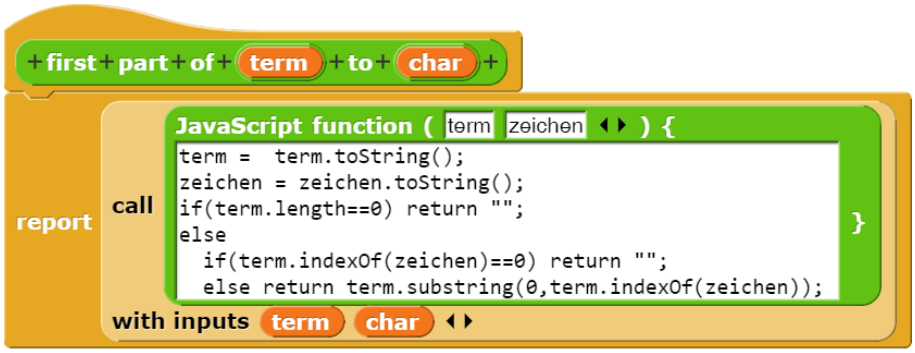


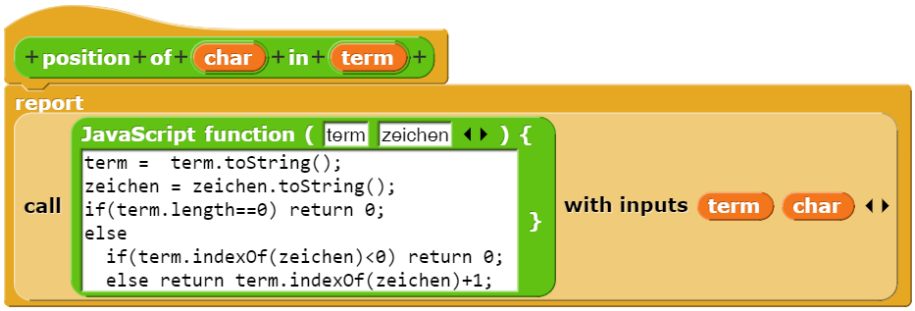
Dieses Verfahren setzen wir für alle Elemente der Sprachdefinition fort. Die Summe besteht entweder aus einem einzelnen Summanden oder einem Summanden, gefolgt vom richtigen Operator (+/-) und einer Summe. Das können wir direkt hinschreiben, wenn wir über ein vorerst noch leeres Prädikat *is <term> a summand?* verfügen.

Wir müssen aufpassen, dass unsere Terme – also Zeichenketten – nicht versehentlich als Zahlen interpretiert werden. Aus diesem Grund haben wir den Typ der Eingabeparameter *term* immer auf „Text“ festgelegt. Vergessen wir das, dann könnte z. B. die Zeichenkette „*123*“ als Zahl *123* interpretiert werden. Das zweite Element der *Zeichenkette* ist z. B. eine *2*, aber in der *Zahl 123* gibt es aber kein zweites Element. Ein entsprechender Zugriff würde zu einem Fehler führen.

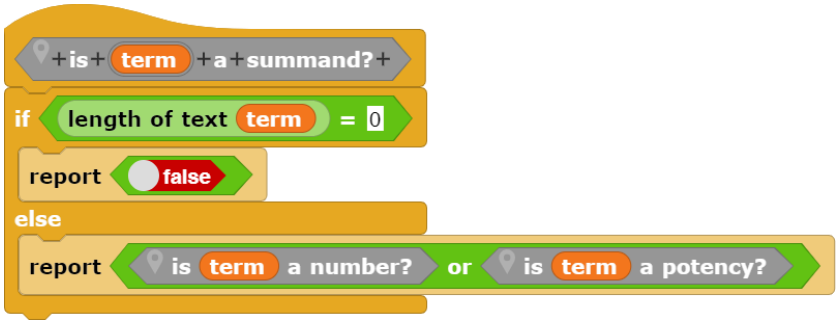
Wie brauchen noch etwas. Der eingegebene Term wird ja nicht mehr insgesamt untersucht, sondern wir müssen ihn ggf. in zwei Teile aufspalten: den *first part of <term> to <zeichen>* und den *rest of <term> from <zeichen>*. Dazu kommt noch die Bestimmung der Position eines Zeichens in einer Zeichenkette: *position of <zeichen> in <term>*. In diesem Fall wollen wir sie als *JavaScript*-Methoden realisieren, weil es etwas auf die Zeit ankommt.[65]





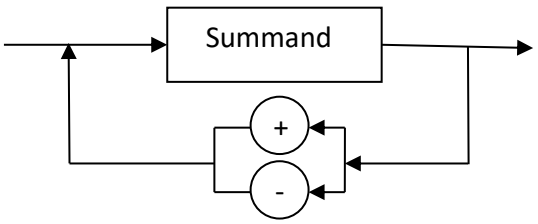


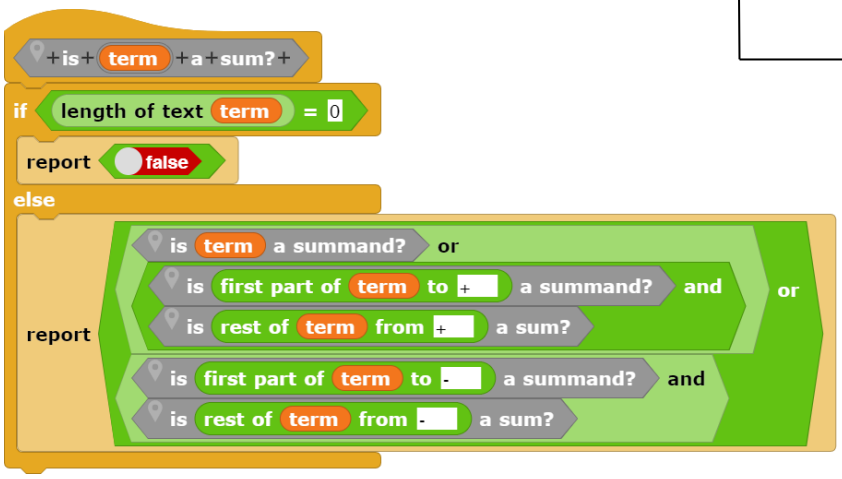
Damit schreiben wir das Prädikat *is <term> a summand?* - mit einer zusätzlichen Sicherheitsabfrage.



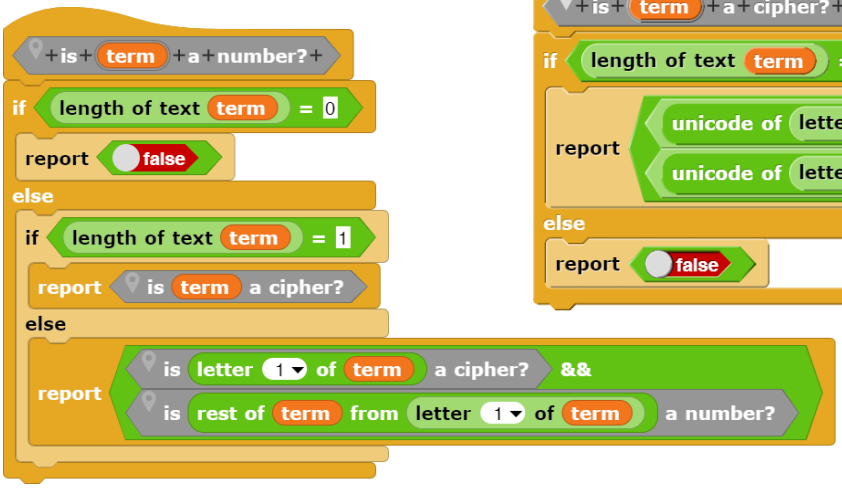
[65] Dafür muss im Settings-Menü die JavaScript-Nutzung explizit zugelassen werden.

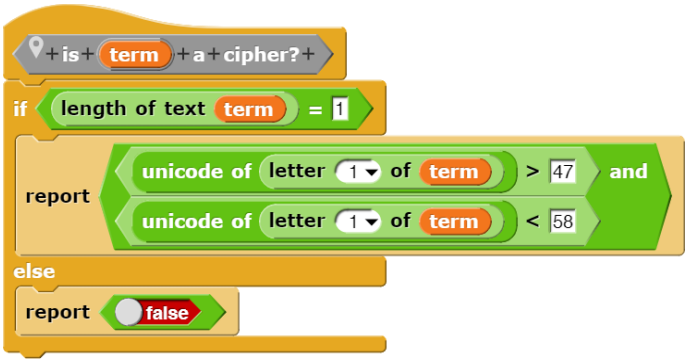
Und jetzt können wir endlich das Prädikat *is <term> a sum?* erstellen.



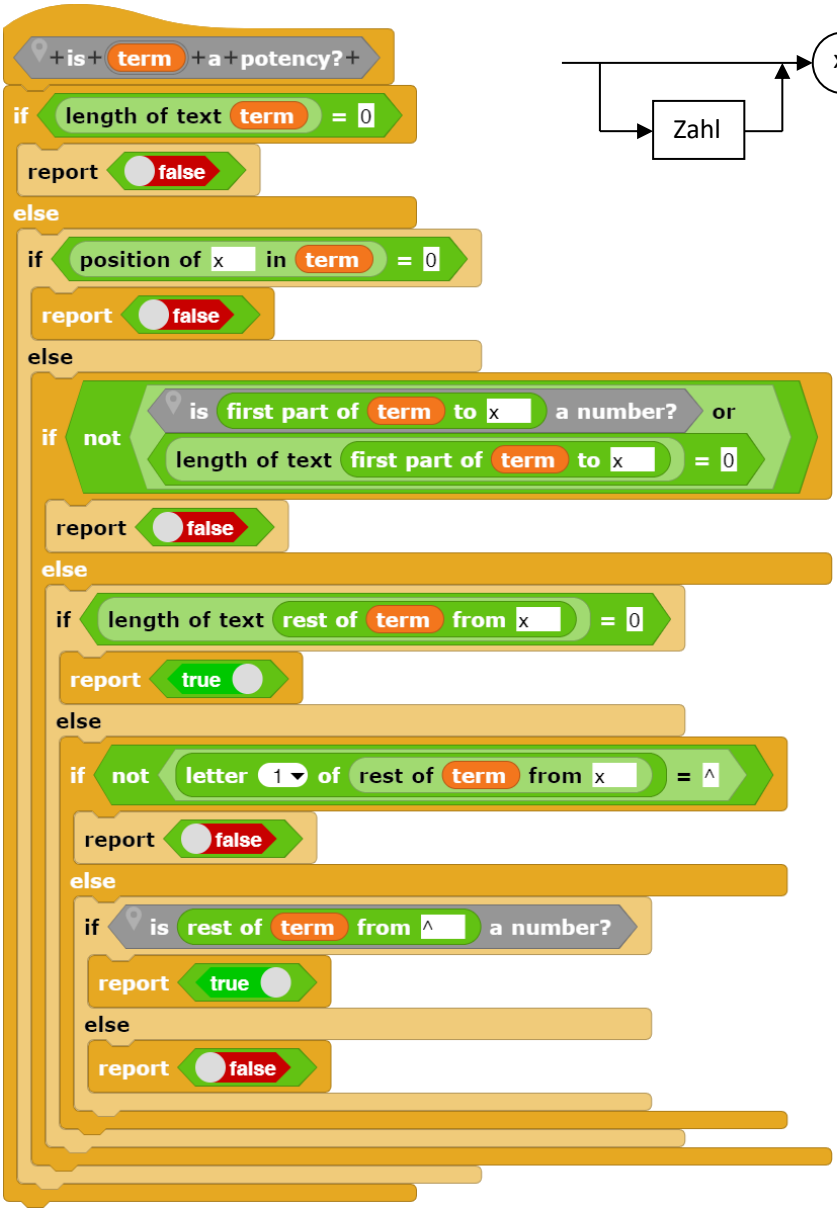


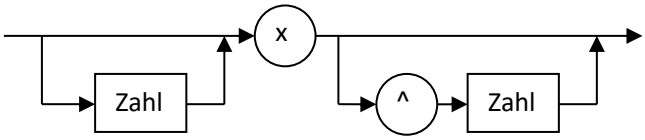
Wir nähern uns dem Ende. *is <term> a number?* ist sehr leicht zu schreiben, wenn man *is <term> a cipher?* kennt:



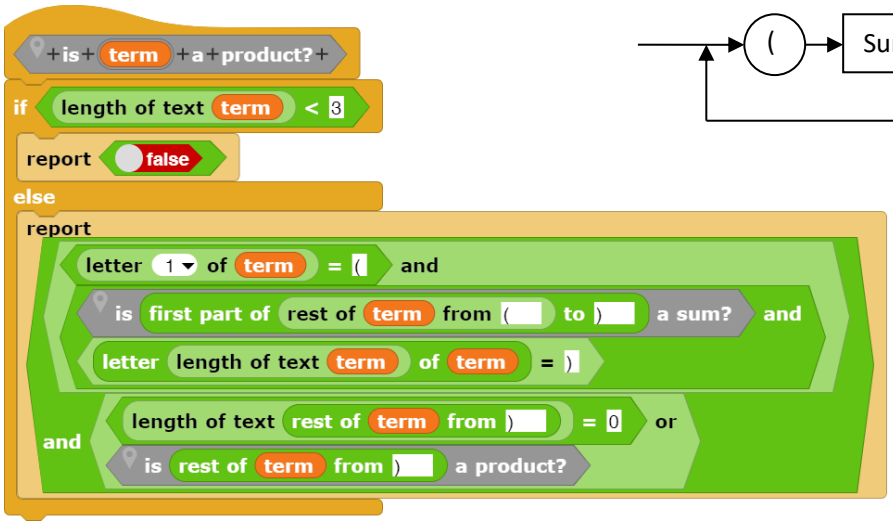


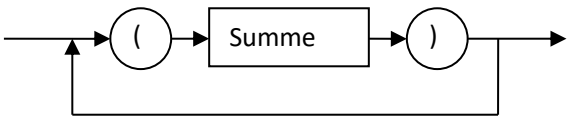
Und wie überprüft man eine Potenz? Das steht ja auch im Syntaxdiagramm – wir müssen nur alle Möglichkeiten abschreiben.





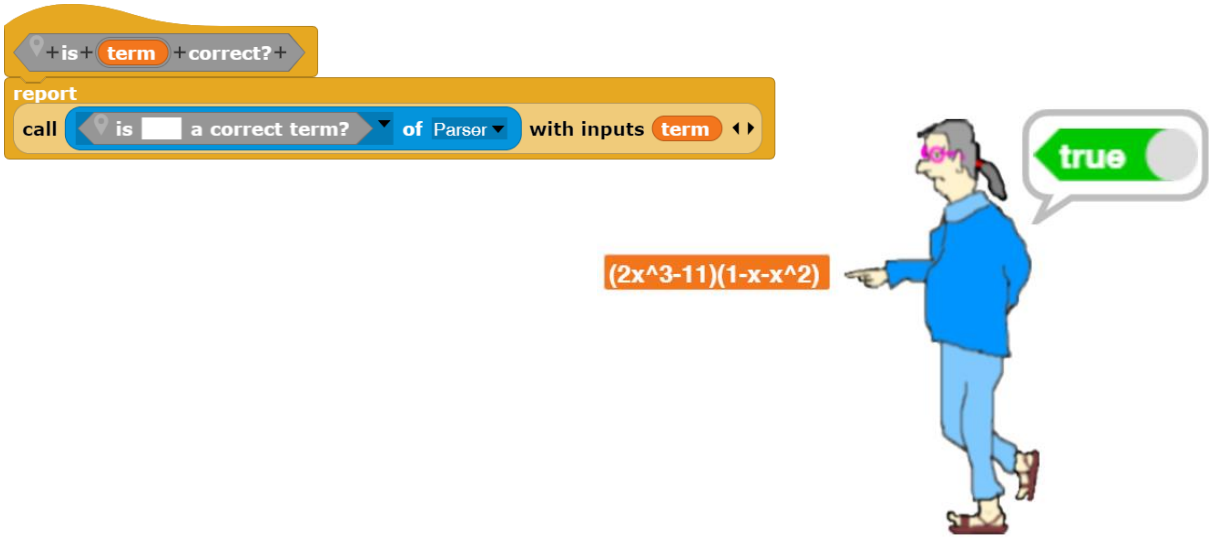
Jetzt fehlt nur noch das Produkt, das sich in direkter Analogie zur Summe formulieren lässt, denn ein Produkt besteht (bei uns) entweder aus einer geklammerten Summe oder einer solchen, gefolgt von einem Produkt.





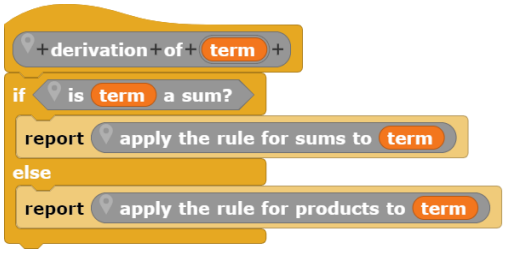
Wir können damit überprüfen („*parsen*“), ob ein eingegebener Term der gewählten Syntax entspricht. Ist das der Fall, dann kann damit weitergearbeitet werden. Unser Mathematiker Gundolf befragt hier den Parser.

Er verpackt diese Abfrage natürlich in einen eigenen Block, um den Anschein zu erwecken, dass er selbst so eine Frage beantworten könnte.



14.3 Funktionsterme ableiten

Wir wollen die erste Ableitung korrekter Funktionsterme bestimmen. Die erforderlichen Methoden sammeln wir beim *Parser*. Da es nur zwei Möglichkeiten für den inneren Aufbau von Termen 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.

Einzelne Summanden abzuleiten ist nicht besonders schwierig.

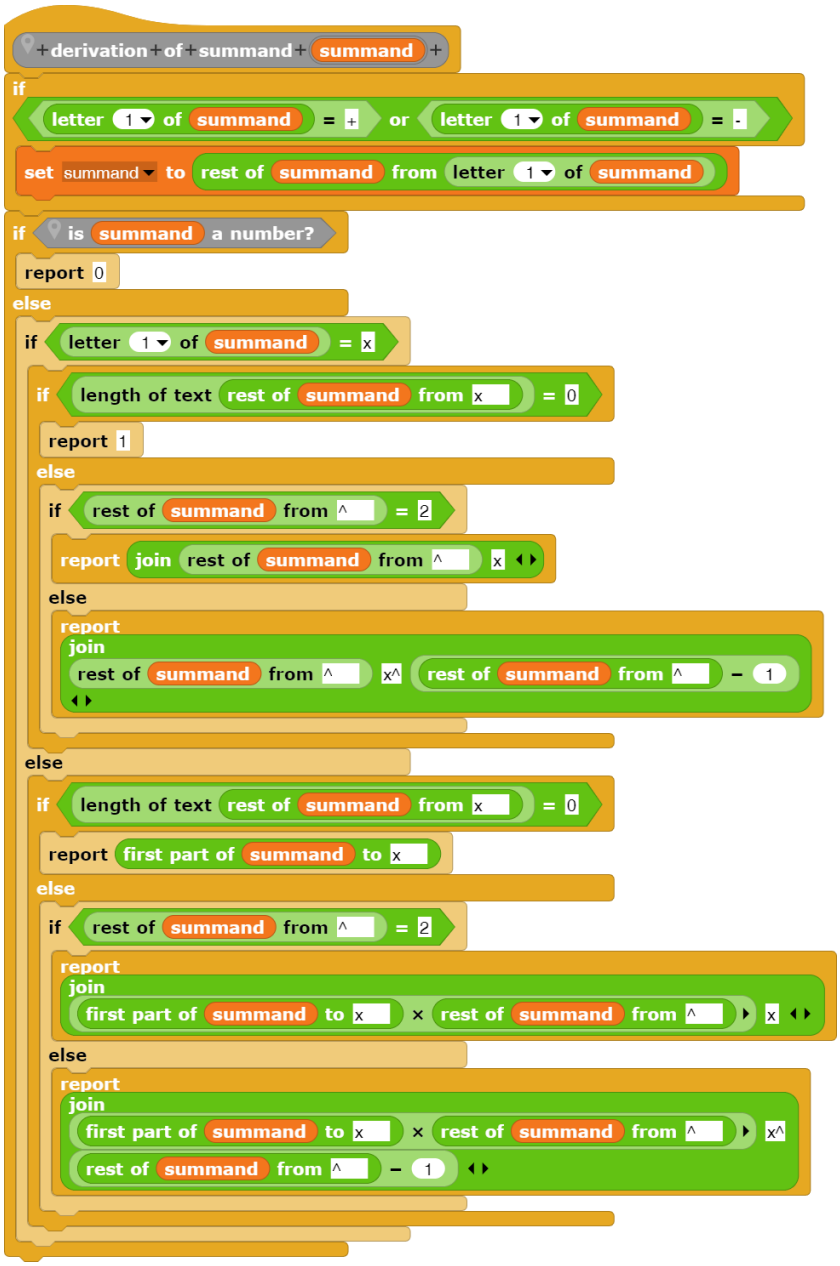
Zahlen ergeben Null.

Die Ableitung von x ist Eins.

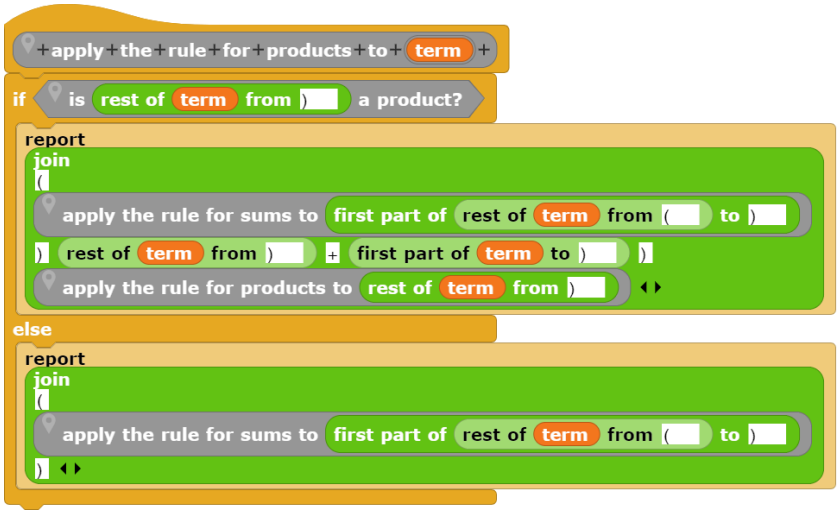
Die Ableitung von x^2 ist 2x,

sonst erhalten wir n*x^(n-1)

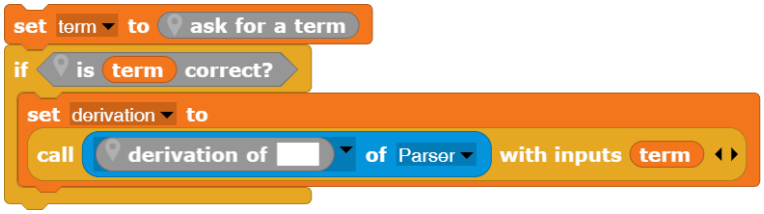
Entsprechend wird ein Faktor vor x berücksichtigt.

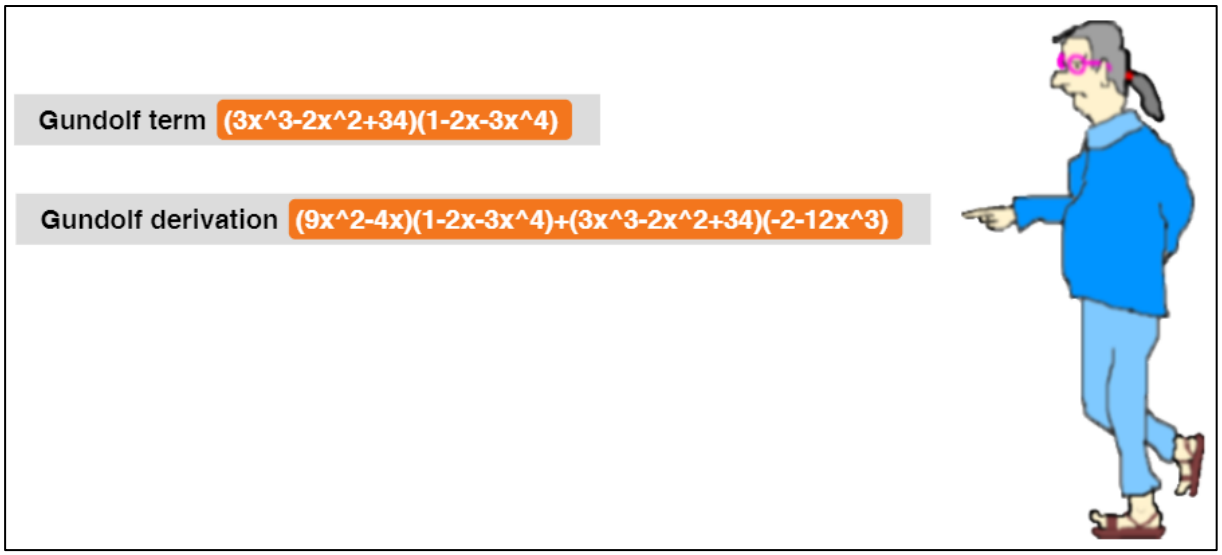


Es fehlt nur noch die Produktregel. Die können wir einfach hinschreiben – unter Hinzufügung einiger Klammern.



Das Ergebnis kann man sogar halbwegs lesen:



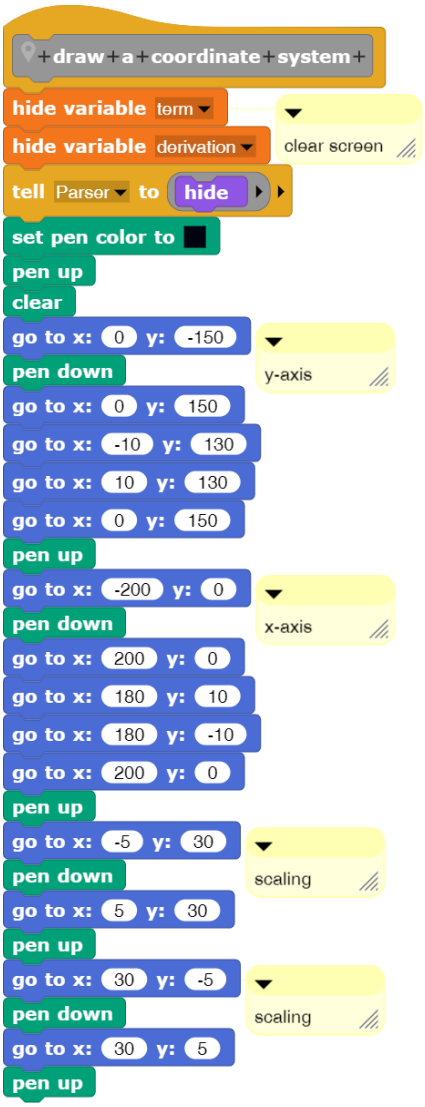


Zu beachten ist, dass die Ableitungen nicht unbedingt unserer stark vereinfachten Definition von Funktionstermen entsprechen und deshalb oft nicht „weiterverarbeitet“ werden können.

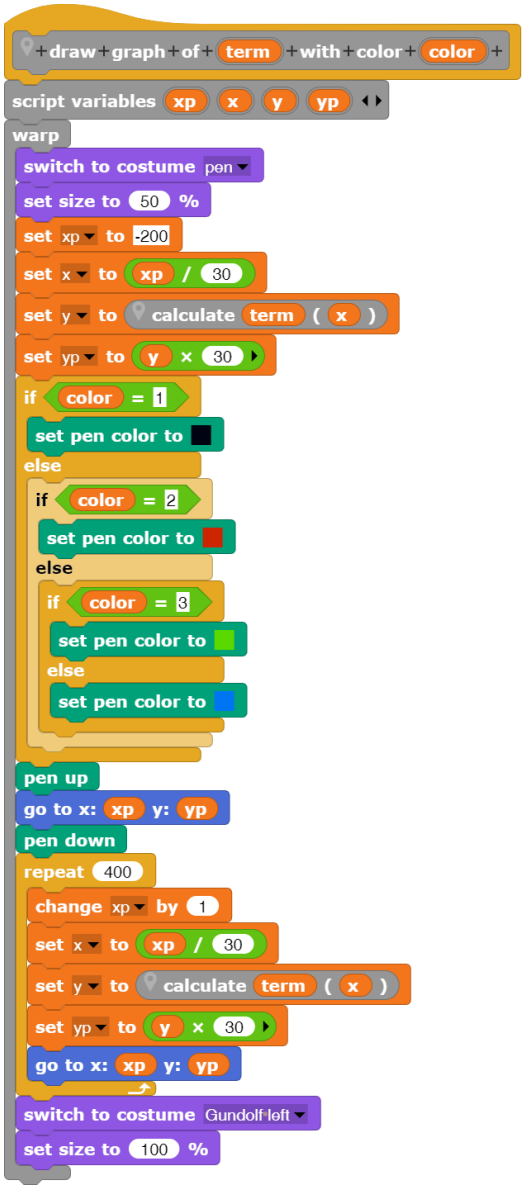
14.4 Funktionswerte berechnen und Graphen zeichnen

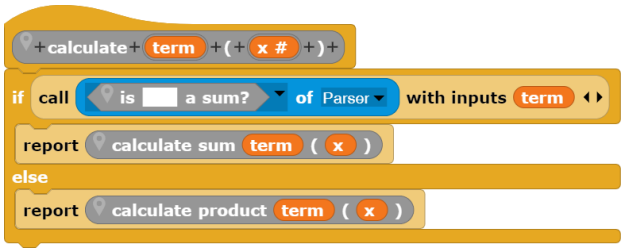
Wenn wir Funktionswerte parsen können, dann können wir sie natürlich auch berechnen. Das Vorgehen ist ganz ähnlich wie beim Parsen, und es wird sehr erleichtert, wenn wir schon wissen, dass der eingegebene Term korrekt ist. Diese Arbeit überlassen wir Gundolf, der bisher ja eigentlich – außer der Selbstdarstellung – ziemlich überflüssig war. Rechnen wird er als Mathematiker ja wohl können!

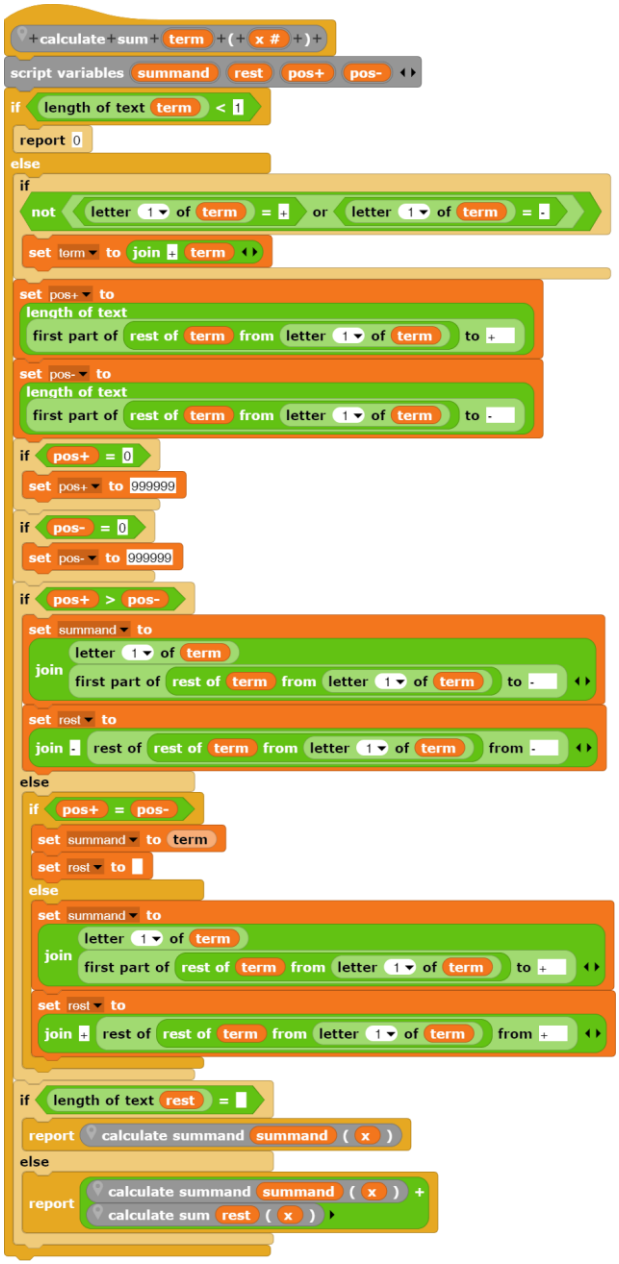
Wir wollen Funktionswerte berechnen und dann die Graphen der Funktion und ihrer ersten Ableitung zeichnen. Dafür muss Gundolf wenigstens einen Graphen zeichnen können.

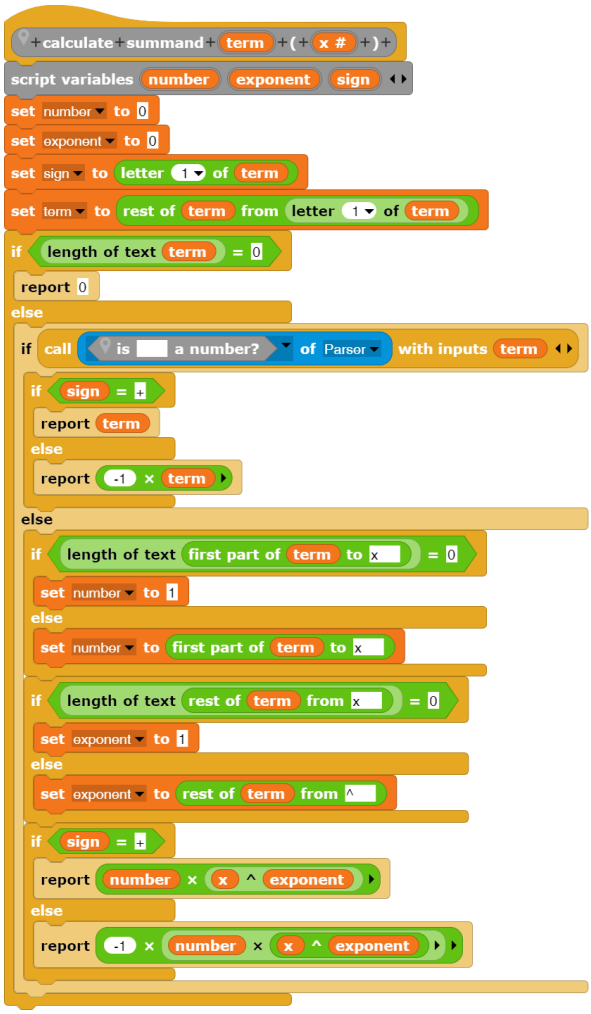


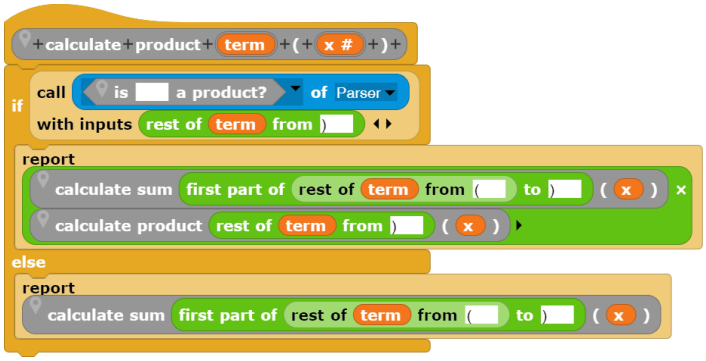
In diesen Skripten sind alle Blöcke schon vorhanden – bis auf einen. Es fehlt noch die Berechnung eines Funktionsterms an der Stelle *x*. Wir geben die entsprechenden Skripte nur an, weil sie sehr ähnlich denen des Parsers sind.




+calculate+ term +(+ x # +)+
if call is a sum? of Parser with inputs term
report calculate sum term (x)
else
report calculate product term (x)

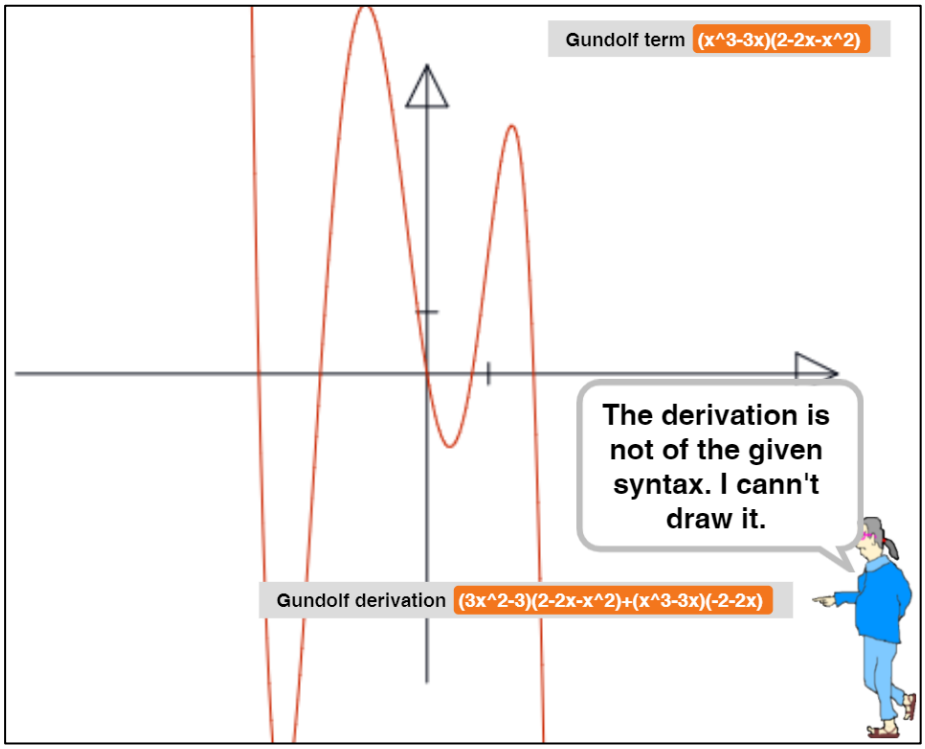

+calculate+sum+ term +(+ x # +)+
script variables summand rest pos+ pos-
if length of text term < 1
report 0
else
if not letter 1 of term = + or letter 1 of term = -
set term to join + term
set pos+ to length of text first part of rest of term from letter 1 of term to +
set pos- to length of text first part of rest of term from letter 1 of term to -
if pos+ = 0
set pos+ to 999999
if pos- = 0
set pos- to 999999
if pos+ > pos-
set summand to join letter 1 of term first part of rest of term from letter 1 of term to -
set rest to join - rest of rest of term from letter 1 of term of term from -
else
if pos+ = pos-
set summand to term
set rest to
else
set summand to join letter 1 of term first part of rest of term from letter 1 of term of term to +
set rest to join + rest of rest of term from letter 1 of term of term from +
if length of text rest =
report calculate summand summand (x)
else
report calculate summand summand (x) + calculate sum rest (x)


+calculate+summand+ term +(+ x # +)+
script variables number exponent sign
set number to 0
set exponent to 0
set sign to letter 1 of term
set term to rest of term from letter 1 of term
if length of text term = 0
report 0
else
if call is a number? of Parser with inputs term
if sign = +
report term
else
report -1 x term
else
if length of text first part of term to x = 0
set number to 1
else
set number to first part of term to x
if length of text rest of term from x = 0
set exponent to 1
else
set exponent to rest of term from ^
if sign = +
report number x x ^ exponent
else
report -1 x number x x ^ exponent


+calculate+product+ term +(+ x # +)+
if call is a product? of Parser with inputs rest of term from)
report
calculate sum first part of rest of term from (to) (x) x
calculate product rest of term from) (x)
else
report
calculate sum first part of rest of term from (to) (x)

Mit deren Hilfe kann Gundolf nun glänzen:

```
when clicked
clear
set size to 50 %
set term to
set derivation to
go to x: 190 y: 0
say
set term to ask for a term
if call is a correct term? of Parser with inputs term
    draw a coordinate system
    draw graph of term with color 2
    set derivation to
        call derivation of of Parser with inputs term
    if
        call is a correct term? of Parser with inputs derivation
        draw graph of derivation with color 3
    else
        say The derivation is not of the given syntax. I cann't draw it.
else
    say That is not a correct term. Try again. for 2 secs
show variable term
show variable derivation
pen up
go to x: 210 y: -110
show
```



14.5 Aufgaben

1. a: Gestalten Sie die Ausgaben etwas **lesefreundlicher**.
   b: Fassen Sie Ergebnisse in der Ableitung so zusammen, dass diese der gegebenen Syntax entspricht und der Graph gezeichnet werden kann.

2. a: Definieren Sie Zahlen mit **Vorzeichen** und ändern Sie die Verarbeitung der Terme entsprechend ab.
   b: Gehen Sie entsprechend für Gleitpunktzahlen (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.

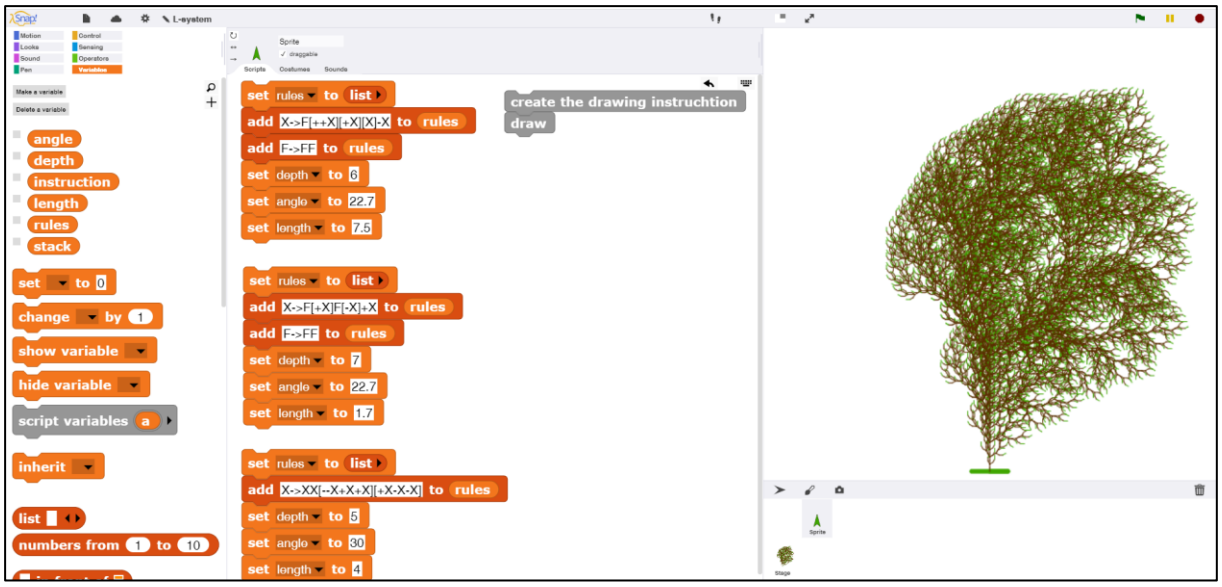
5. Lassen Sie Funktionsterme zu, die die Anwendung der **Kettenregel** erfordern. Implementieren Sie entsprechende Prädikate und Zeichenkettenfunktionen.

6. a: Lassen Sie die Graphen auch der anderen Funktionsarten zeichnen, nachdem sie geparst wurden.
   b: Lassen Sie eine Auswahl der zu zeichnenden Graphen (Funktion, erste und zweite Ableitung) zu.

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.

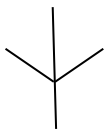
15 Künstliche Pflanzen: L-Systeme

Altersstufe: *Sekundarstufe II* Material: *L-systems*



15.1 L-Systeme

In den Systemen nach Aristid Lindenmayer[66] werden Pflanzen durch ein Regelsystem beschrieben, das aus einem *Axiom*, einem ersten Zeichen, durch Ersetzungen die Zeichenanweisung für eine *Turtle* erzeugt. Man kann sich das so vorstellen, dass – ausgehend von einem Spross – die Pflanze bis zur nächsten Verzeigung gezeichnet wird. Deren Position wird auf einem *Stapel* (*stack*) abgespeichert, dann werden die Zweige nacheinander beschrieben, wobei nach jedem Ast zur Verzweigung zurückgekehrt wird. Die Turtle beherrscht nur die Vorwärtsbewegung (*F*) sowie Drehungen um einen festen Winkel (+ und -). Das Speichern der Turtleposition und -richtung sowie die Wiederherstellung dieses Zustands wird durch eckige Klammern (*[* und *]*) symbolisiert. Ein einfacher Trieb mit einer Dreifachverzweigung lässt sich beschreiben durch



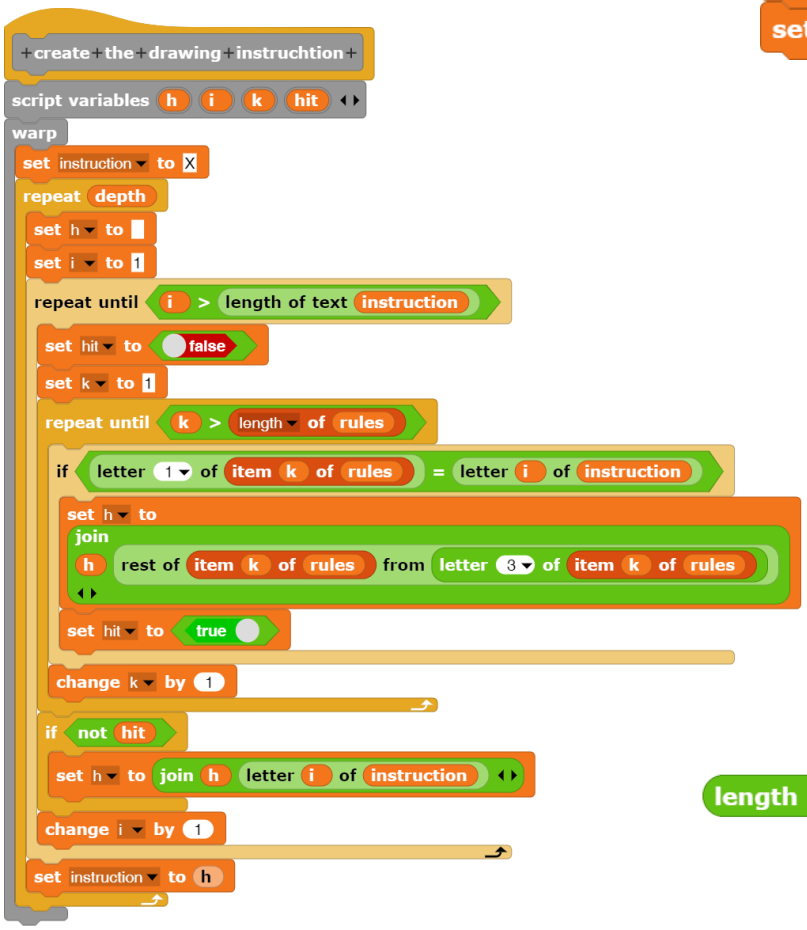
Axiom: *X* Regel: *X* → *F[-X][+X]FX*

Wird diese Regel mehrfach angewandt, dann kann die Pflanze an den Positionen wachsen, an denen ein *X* eingefügt wurde. Damit die älteren Teile der Pflanze mitwachsen, wird oft eine Regel *F* →*FF* eingefügt.

[66] https://de.wikipedia.org/wiki/Lindenmayer-System

15.2 Die Zeichenanweisung erzeugen

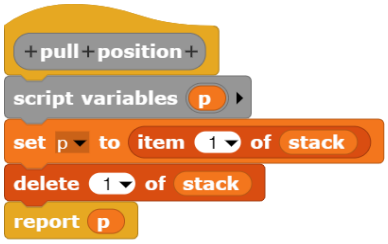
Zuerst einmal brauchen wir ein Regelsystem, also eine Listen-Variable *rules*, der die gewünschten Regeln zeilenweise als Zeichenketten hinzugefügt werden. Das zu ersetzende Zeichen steht ganz vorne, dann folgen „->“ und die Ersetzung ab Zeichen 4. Die Rekursionstiefe, der vorgegebene Winkel und die Länge der Zeichenstrecke werden ebenfalls Variablen zugewiesen.



Bei der Erzeugung der Zeichenanweisung starten wir mit dem *Axiom X*. Dann erzeugen wir einen Hilfsstring *h*, in dem die Ersetzungen pro Durchlauf durchgeführt werden: immer, wenn ein zu ersetzendes Zeichen in der alten Zeichenanweisung gefunden wurde, fügen wir die Ersetzung an *h* an. Zum Schluss ersetzt *h* die Zeichenanweisung und der nächste Ersetzungsdurchgang wird gestartet. Das Ergebnis kann ganz schön lang werden!

15.3 Die Stapeloperationen

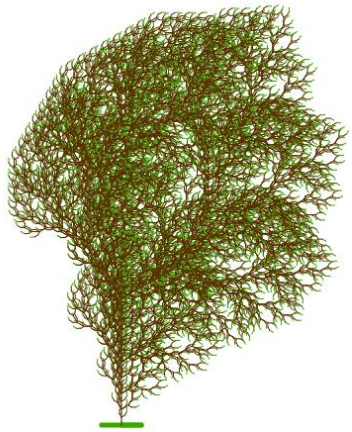
Als Stapel zur Ablage der Turtlepositionen benutzen wir eine einfache Liste. Operationen werden nur am Anfang der Liste ausgeführt – schon haben wir einen Stapel (*stack*). Das Ablegen wird üblicherweise durch eine Operation *push* erledigt. Wir speichern eine dreielementige Liste mit *x*- und *y-Position* sowie die momentane *Richtung*. Durch *pull* wird die zuletzt gespeicherte Position zurückgeholt und aus der Liste entfernt.

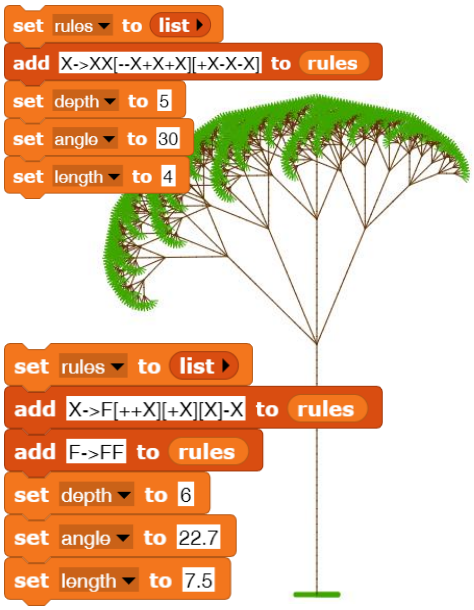


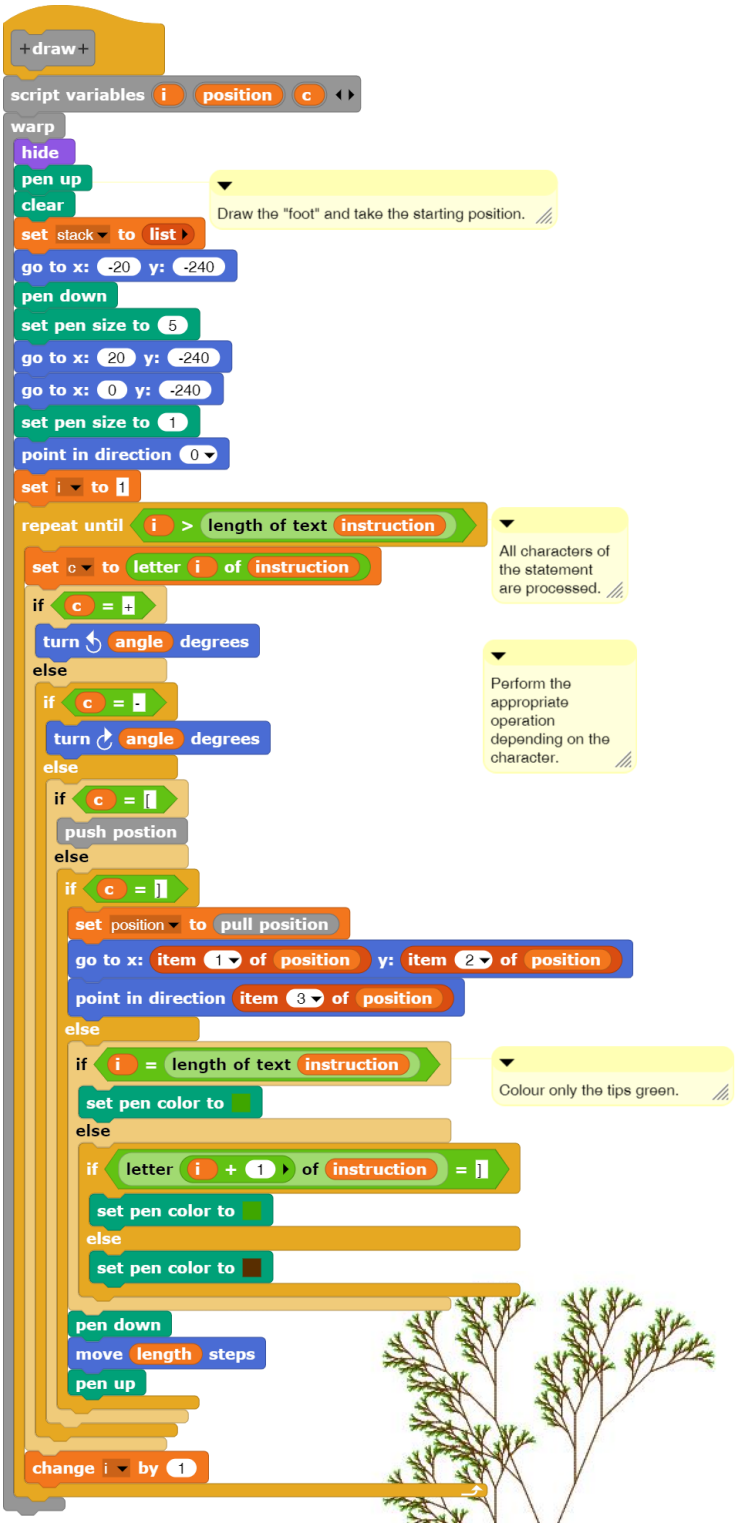
15.4 Das Zeichnen der Pflanzen

Das Zeichnen der Pflanzen ist sehr einfach, da alle unsere Sprites als Turtle benutzt werden können. Wir vergrößern unsere Arbeitsfläche auf 500x500 (im Einstellungsmenü *stage size…* wählen) und lassen die Turtle den „Fuß“ zeichnen, auf dem die Pflanze wächst. Danach wird die Zeichenkette mit den Zeichenanweisungen zeichenweise durchlaufen, wobei für jedes Zeichen die entsprechende Turtle-Operation oder eine Stack-Operation ausgeführt wird. Als kleines Schmankerl zeichnen wir die „Spitzen“ der Pflanze grün. (*Spitzen erkennt man daran, dass als Nächstes zur letzten Turtleposition zurückgegangen wird, also eine pull-Operation folgt.*)

Beispiele:







15.5 Aufgaben

1. a: Suchen Sie im Netz nach **Grammatiken** für L-Systeme. Erzeugen Sie die entsprechenden Pflanzen.
   b: Wählen Sie eine Pflanzenart, z. B. eine bestimmte Baumart, aus und studieren Sie deren Bau gründlich anhand von Bildern. Achten Sie besonders auf Wachstumsstellen. Beschreiben Sie dann deren Aufbau anhand einer L-System-Grammatik. Überprüfen Sie das Ergebnis mithilfe des Programms.

2. a: Weshalb sind die bisher betrachteten Grammatiken eigentlich „**kontextfrei**“? Was bedeutet das für die erzeugten Pflanzen?
   b: Informieren Sie sich im Netz, ob auch andere als kontextfreie Grammatiken für die Beschreibung künstlicher Pflanzen benutzt werden. Falls ja: weshalb eigentlich?

3. a: Im Programm wurden die Spitzen der Äste (als „Blätter“) grün gefärbt. Ersetzen Sie diese grünen Stücke durch **schönere Blätter**.
   b: Übertragen Sie das Prinzip auf das Zeichnen der Dicke der Äste. Lassen Sie sich ruhig etwas einfallen! 😉

4. Pflanzen wachsen ja nicht immer gleich: es gibt Stürme, tobende Kinder, Hobbygärtner, Wetterkatastrophen, … Bringen Sie etwas **Zufall** ins Spiel, um unterschiedlich geformte Pflanzen des gleichen Typs zu erzeugen.

5. a: Die **Stapeloperationen** wurden immer am Anfang der Liste ausgeführt. Könnte man auch das Ende nehmen? Falls ja: weshalb?
   b: Würde sich etwas ändern, wenn man am Anfang des Stapels einfügt und am Ende die Positionen entfernt? Falls ja: weshalb?

6. Die Benutzer des L-System-Programms können als Grammatik ja sonstwas eingeben. Überprüfen Sie deren Eingaben durch einen **Parser**, bevor sie versuchen, die Pflanze zu erzeugen.

7. a: Wie wären die Regeln für L-Systeme zu ändern, wenn wir **dreidimensionale** Pflanzen erzeugen wollten? Was bedeutete das für das Zeichnen der Pflanzen? Gibt es Turtles für dreidimensionales Zeichnen?
   b: Informieren Sie sich im Netz über Gebiete, in denen künstliche Pflanzen benutzt werden.

8. Zeichnet man auch künstliche **Tiere**? Künstliche Menschen? Falls ja: wo? Wie macht man das?

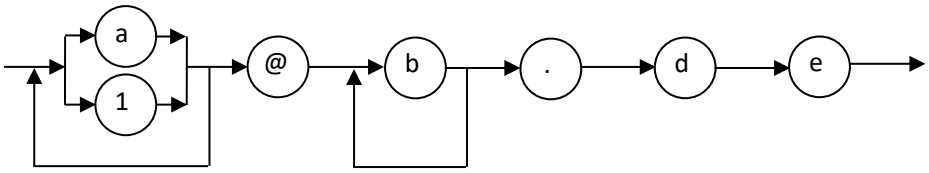
16 Automaten

16.1 Korrekte Mailadressen

Altersstufe: *Sekundarstufe I/II* Material: *Email addresses*

Wir wollen mithilfe eines *endlichen Automaten* (*EA*) überprüfen, ob eine Mailadresse korrekt ist. Dafür müssen wir natürlich erstmal wissen, was „korrekt“ bedeutet. Wir geben ein Syntaxdiagramm an:

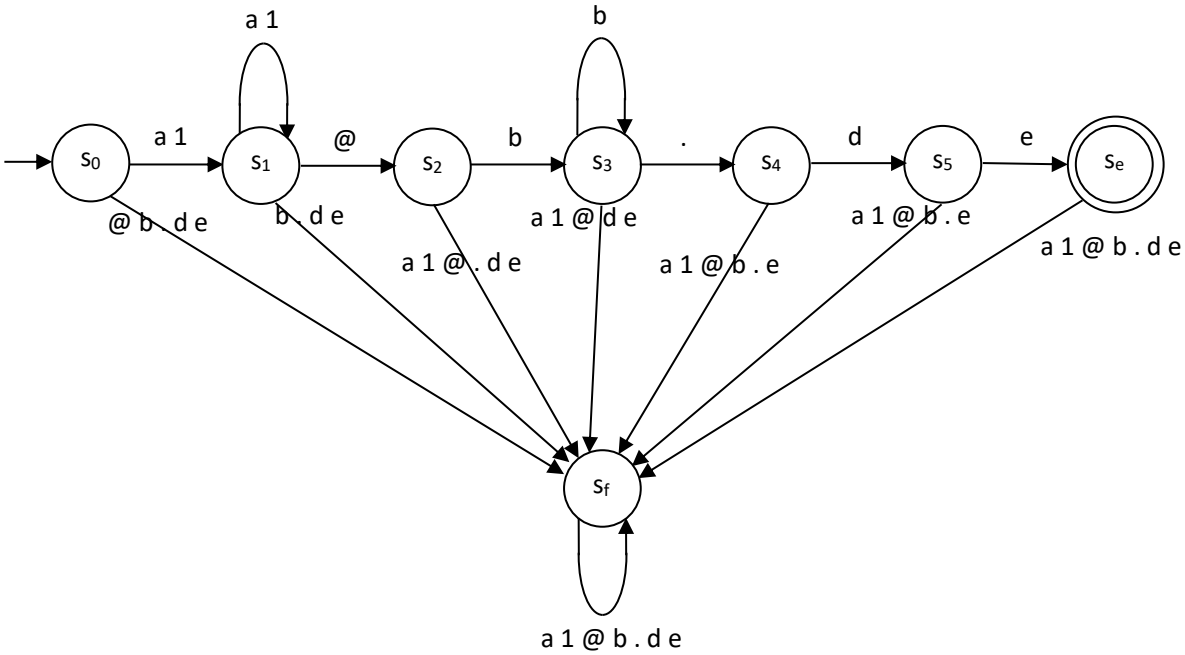
Mailadresse:



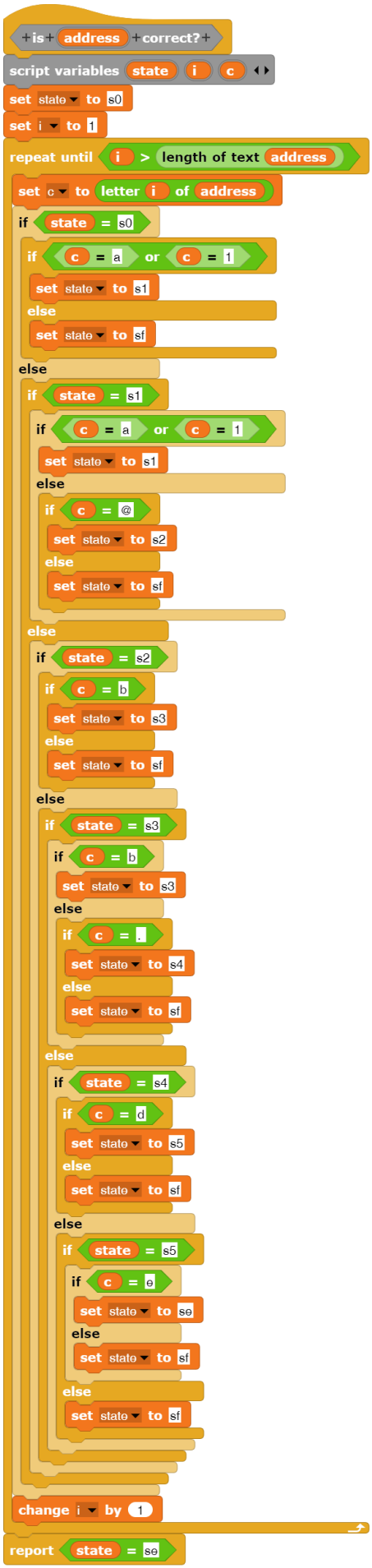
In dieser vereinfachten Form bestehen die Teilnehmernamen also aus den Zeichen *a* und *1* (als Stellvertreter für Buchstaben und Sonderzeichen) in bunter Reihenfolge, dann folgt das übliche @. Der Mailservername besteht nur aus *b*s, und – abgetrennt durch den Punkt – folgt als Domainname *de*.

Korrekte Email-Adressen sind z. B. a@b.de a1a@bbb.de , falsch wäre z. B. 1@c.com.

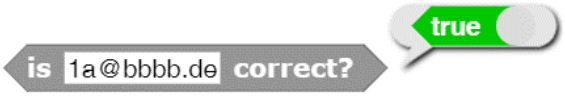
Übersetzt in einen endlichen Automaten erhalten wir dessen Zustandsdiagramm (die Eingabezeichen werden an den Kanten des Graphen mit Leerzeichen als Trennzeichen aufgezählt):



Die Übersetzung in ein *Snap!*-Skript kann gut als Prädikat erfolgen, denn die Antwort des Automaten besteht in *true* (der Endzustand s_e wurde erreicht) oder *false* (ein anderer Zustand wurde erreicht, typischerweise der Fehlerzustand s_f). Im Skript wird die überprüfte Adresse zeichenweise durchlaufen. Ausgehend vom Anfangszustand s_0 wird überprüft, ob das aktuelle Zeichen zulässig ist. Ist das der Fall, dann wird in den im Zustandsdiagramm angegebenen Folgezustand gewechselt, sonst in den Fehlerzustand. Das Skript ist zwar recht lang, besteht aber nur aus geschachtelten Alternativen, die eine direkte Übersetzung des Zustandsdiagramms darstellen.



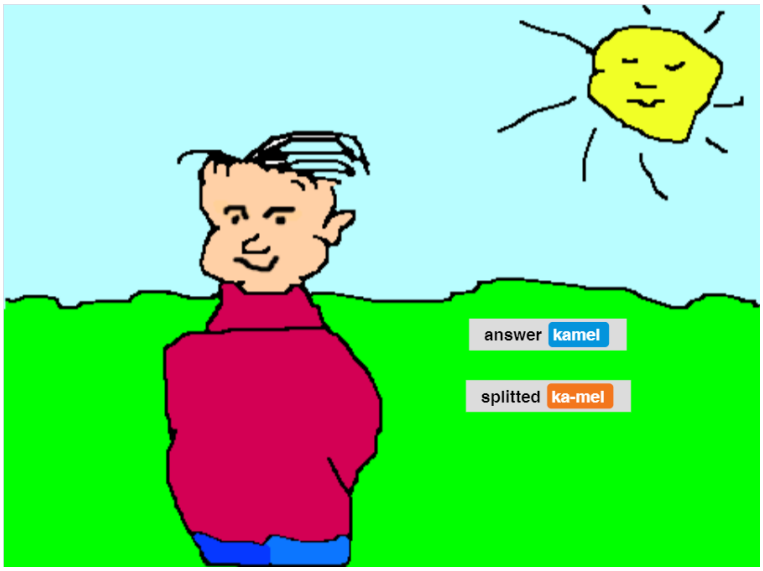
Bei der Überprüfung der Mailadressen kann dann das erstellte Prädikat eingesetzt werden.



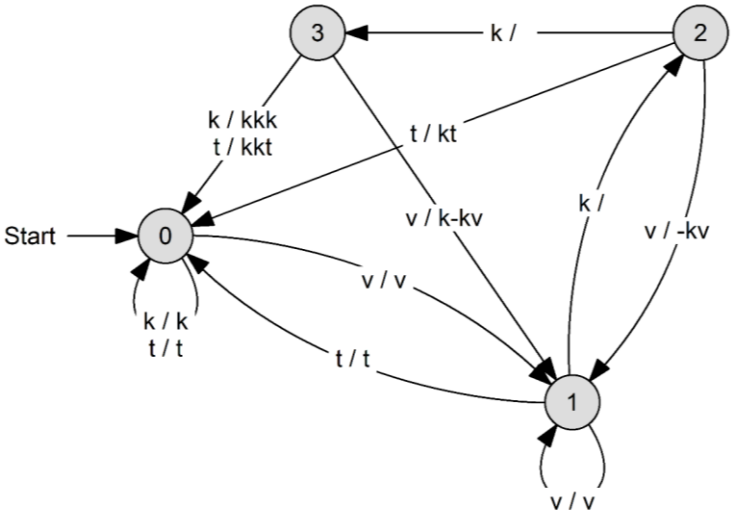
16.2 Silbentrennung: Kevin spricht[67]

Altersstufe: *Sekundarstufe II* Material: *Kevin speaks*

Mealy-Automaten kann man dazu benutzen, eine einfache Silbentrennung zu implementieren, die überraschend gut funktioniert. Zusätzlich wollen wir das Sprite *Kevin* dazu bewegen, die eingegebenen Worte auszusprechen. Das zweite hört sich schwieriger an, als es ist: wenn wir die Silben haben, dann können wir für jede Silbe ein Bild mit der Mundstellung erzeugen, dessen Name der Silbe entspricht (z. B. *AU.png*) und dazu die gesprochene Silbe aufnehmen (z. B. als *AU.wav*). Diese Dateien ziehen wir in die *Costumes*- bzw. *Sounds*-Bereiche von *Snap!* und rufen sie von dort auf.



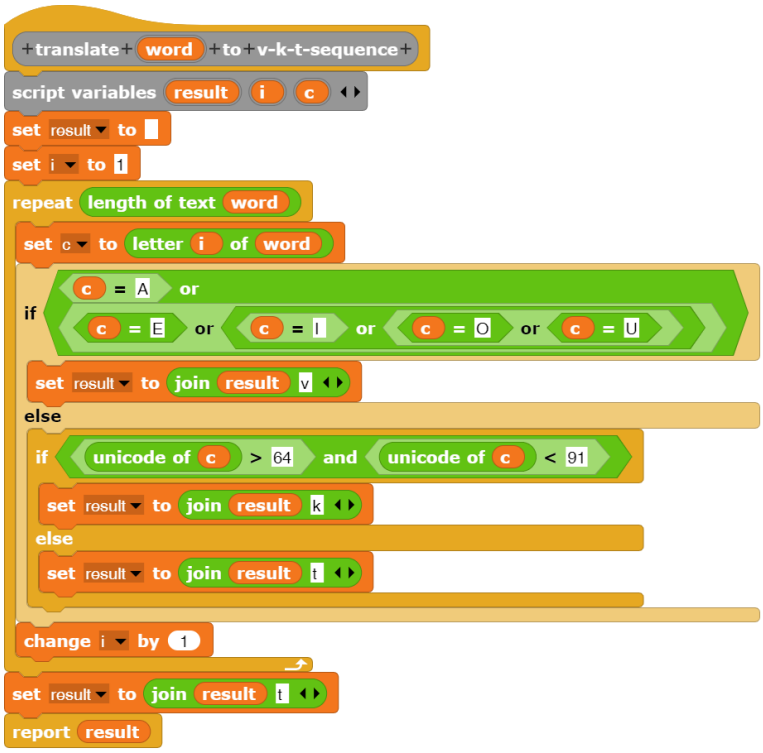
Wir gehen von dem nebenstehenden, sehr einfachen Mealy-Automaten aus. Dessen Eingabealphabet besteht aus Vokalen (*v*), Konsonanten (*k*) und sonstigen Trennzeichen (*t*). Er fügt einige Trennzeichen für die Silbentrennung ein, arbeitet dabei aber natürlich unvollständig und teilweise falsch. Er trennt die Zeichenfolgen *vkv* in *v-kv* und *vkkv* in *vk-kv*.

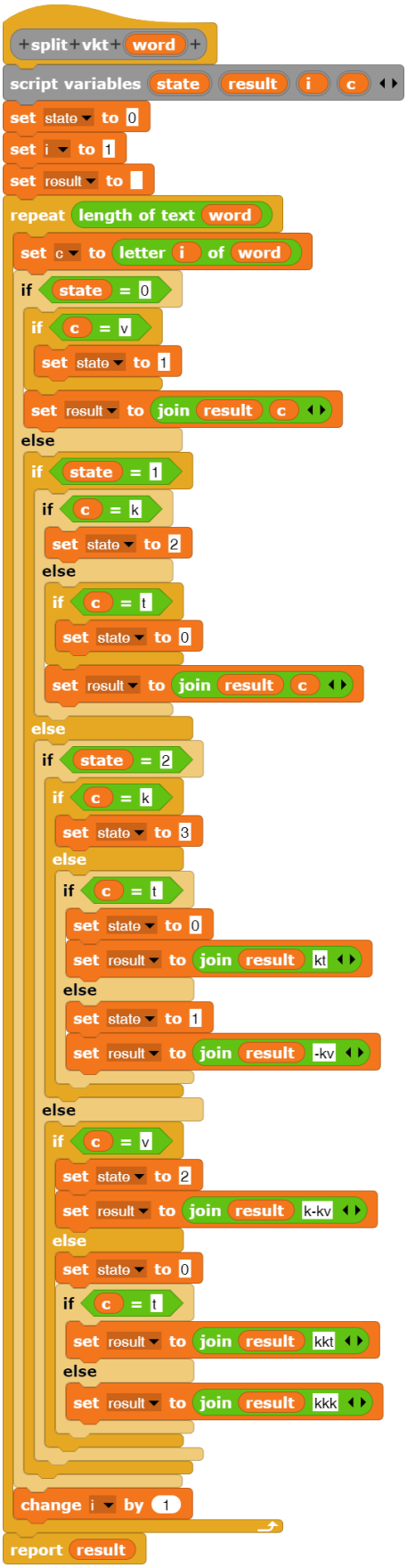


Mithilfe von *ask and wait* geben wir Worte ein, die dann getrennt werden. Da sich Benutzer von Programmen nie an die Vorgaben halten, sorgen wir zuerst einmal dafür, dass nur Großbuchstaben im Wort vorkommen. Dazu müssen wir zumindest ein einzelnes Zeichen ggf. in Großschrift verwandeln können. Die Funktion dafür haben wir schon bei der Vigenère-Verschlüsselung geschrieben, ebenso wie die für die Umwandlung ganzer Worte.

Ein in Großschrift verwandeltes Wort kann auf ähnliche Weise in eine Folge aus den Zeichen *v*, *k* und *t* transformiert werden. Die Vokale sind sehr einfach zu finden, die Konsonanten sind Buchstaben, die keine Vokale sind, der Rest wird als Trennzeichen behandelt. Aus praktischen Gründen wird zuletzt noch ein *t*-Zeichen angehängt. Damit ist immer wenigstens ein Zeichen vorhanden – und wir gelangen beim Automaten immer zuletzt in den Zustand 0.

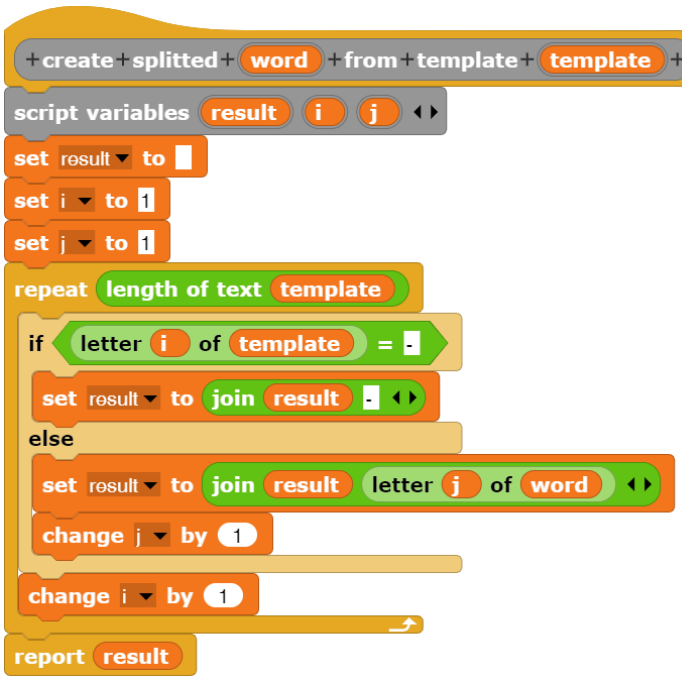
[67] Nach einer Idee von Wilfrid Herget.



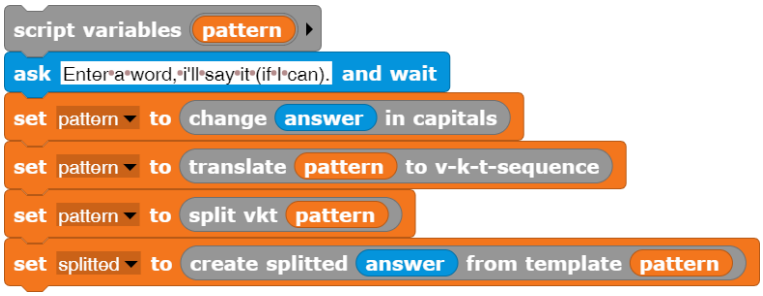


Jetzt wird getrennt. Wir lesen Zeichen für Zeichen der Folge aus den Zeichen *v*, *k* und *t* und schreiben unseren Automaten ab: Je nach Zustand wird angegeben, welcher Folgezustand eingenommen wird und welche Zeichen ausgegeben werden.

Zuletzt müssen wir die *vkt*-Folge wieder in die ursprünglichen Zeichen zurück verwandeln – mit den Trennzeichen dazwischen. Dazu durchlaufen wir die *vkt*-Folge mit den Trennzeichen (Index: *i*) als *template* und bauen die Ergebnisfolge aus den Zeichen des eingegebenen Wortes auf (Index *j*). Wir ändern *j* allerdings nur, wenn *i* nicht auf ein Trennzeichen (-) im Muster zeigt.

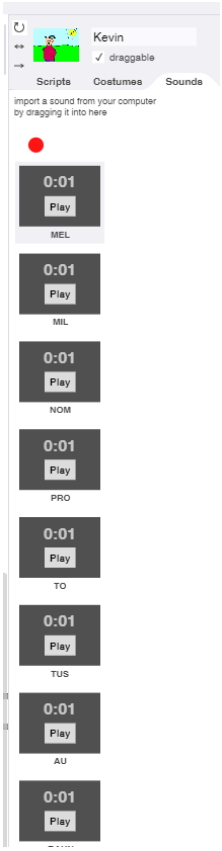


Diese Funktionen können wir nun schrittweise nacheinander benutzen, um ein Wort zu trennen:



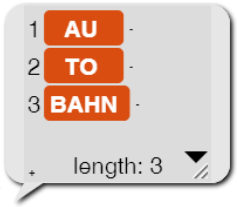
Und natürlich können wir solche Anweisungsfolgen in einem neuen Block bündeln.

Die in Silben zerlegte Worte sollen vom Computer ausgesprochen werden, ähnlich wie in Navigationssystemen, automatischen Zeitansagen oder anderen „Computerstimmen“. Wenn wir Silben statt ganzen Worten speichern, dann benötigen wir wesentlich weniger Speicherplatz, weil sich die Silben mehrfach verwenden lassen. (Schöner wird es dadurch aber nicht!)

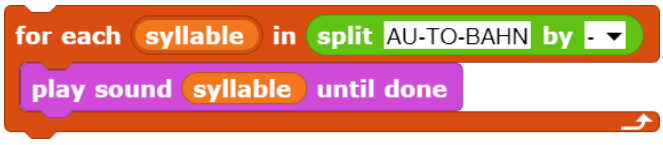


Zuerst wählen wir einige Worte: *Autobahn, autonom, Automat, Pronom, Promille, Kamille, Kamel, Kaktus*. Deren Silben nehmen wir jeweils im Recorder des *Sounds*-Bereichs auf und ändern ihren Namen auf den der Silbe in Großbuchstaben.

Da die eingegebenen Worte mit Trennzeichen versehen wurden (s.o.), erhalten wir (so in etwa) die Silben, wenn wir das Wort „zerlegen“. Dafür stellt *Snap!* den Befehl *split by* bereit. Der Block erzeugt eine Liste von Wortteilen. Geben wir *AU-TO-BAHN* ein und trennen beim Zeichen „-“, dann erhalten wir:



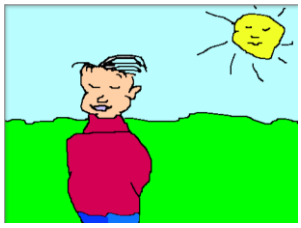
Da unsere Sound-Dateien genauso heißen wie die Silben, können wir sie mit *play sound until done* abspielen, indem wir die Silbe als Eingabeparameter des Blocks wählen.



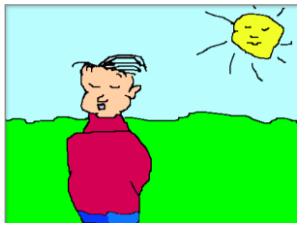
Wir können also den Computer Worte aussprechen lassen, indem wir

- das eingegebene Wort trennen,
- und in seine Silben zerlegen,
- und die Silben dieser Liste nacheinander „aussprechen“ lassen.

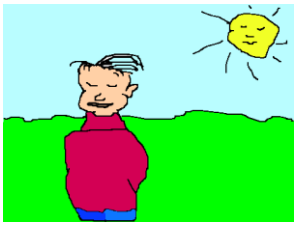
Zu den verschiedenen Silben zeichnen wir jeweils ein Kostüm für Kevin.



AU



TO



BAHN

Diese Kostüme zeigen wir während des Sprechens der Silben an.

Das Aussprechen von Worten erfolgt dann über den Aufruf dieses Skripts mit den entsprechenden Silben.

16.3 Gekoppelte Turingmaschinen[68]

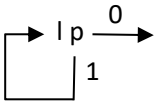
Altersstufe: *Sekundarstufe II* Material: *Coupled Turing adder*

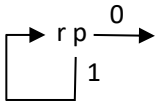
Beschreibt man Turingmaschinen durch Zustandsgraphen, dann scheint den Lernenden die Bedeutung, die diesem Modell zugewiesen wird, stark übertrieben, denn die Probleme, die sich durch einen noch lesbaren Graphen beschreiben lassen, sind denn doch ziemlich übersichtlich. Wesentlich mächtigere Werkzeuge lassen sich im Modell der gekoppelten Turingmaschinen erzeugen, bei denen der Anfangszustand der nächsten Maschine dem Endzustand ihrer Vorgängerin entspricht. Aus sehr einfachen Systemen lassen sich dann immer leistungsfähigere Konstruktionen erstellen. Es entsteht eine Art Makro-Sprache, in der Themen der Berechenbarkeit und Entscheidbarkeit formulierbar, vor allem aber enaktiv erfahrbar sind.

Unser System von elementaren Turingmaschinen arbeitet auf einem *Turingband*, das nur Einsen und Nullen enthält. Die Nullen dienen als Trennzeichen, sodass Zahlen durch Einsenfolgen darzustellen sind. Die Zahl *n* wird entsprechend durch *n+1* Einsen kodiert, damit auch die Null einen Code hat. In der *Standardlage* befindet sich der Kopf der Turingmaschine über der am weitesten rechts stehenden Eins. Alle Einsengruppen müssen durch genau eine Null getrennt sein und am linken Rand des Bandes stehen zwei Nullen. Nach der Arbeit befindet sich die Maschine wieder in der Standardlage. Aus dieser beginnt dann die nächste Maschine zu arbeiten.

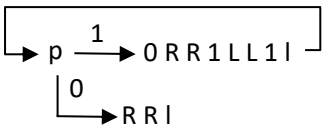
Als elementare Maschinen stehen die *1-* und die *0-Maschine* zur Verfügung, die das entsprechende Zeichen an der Kopfposition auf das Band schreiben. Sonst tun sie nichts. Die *kleine Linksmaschine l* verschiebt den Kopf der Turingmaschine um eine Position nach links, die *kleine Rechtsmaschine r* nach rechts. Zusätzlich gibt es eine *Prüfmaschine p*, die prüft, welches Zeichen an der aktuellen Kopfposition vorliegt. Je nach Ergebnis verzweigt sie in einen von zwei Zuständen, an die dann weitere Maschinen angekoppelt werden können. Das war es schon.

Weil sie oft benötigt werden, entwerfen wir zwei neue Maschinen, die *große Linksmaschine L*, die nach links über eine Einsengruppe hinwegläuft, und entsprechend eine *große Rechtsmaschine R*. Diese können wie folgt realisiert werden:

L: 

R: 

Die Kopiermaschine *K1* kopiert eine Einsengruppe nach rechts.

K1: 

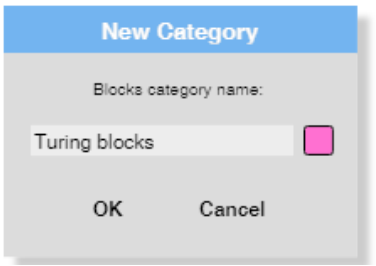
Wenn die Kopiermaschine *K2* eine Einsengruppe über eine zweite hinüber nach rechts kopiert, dann können wir mithilfe eines Turingaddierers *A* schon mal Summen berechnen.

A: *K2 K2 L 1 R l 0 l 0 l*

Probieren Sie es aus!

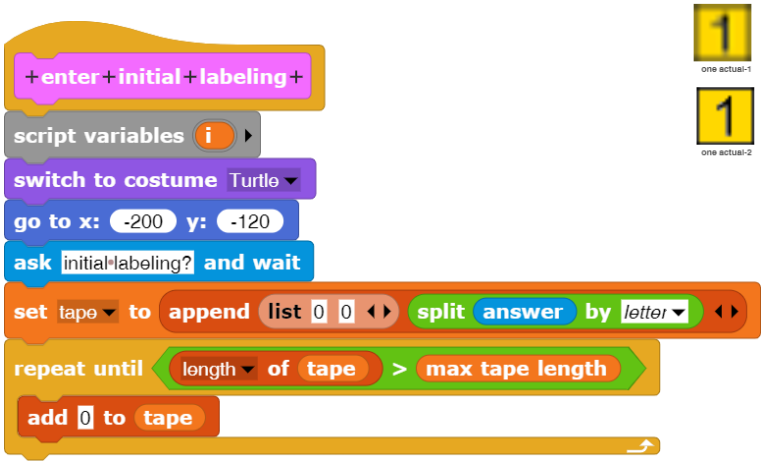
[68] nach Eckart Modrow, Theoretische Informatik mit Delphi, emu-online, 2005

Statt die Maschinen auf dem Papier zu testen, soll eine Makrosprache entwickelt werden, mit der sich unsere gekoppelten Turingmaschinen erzeugen lassen. Da wir nur neu zu entwickelnde *Turing blocks* benutzen wollen, führen wir dafür eine neue Kategorie ein – in geschmackvollem Pink.

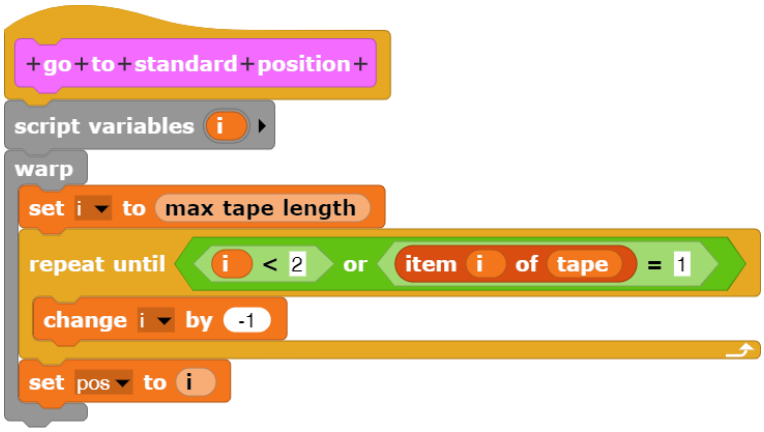


Für die Simulation der Maschinen benötigen wir ein Arbeitsband, das aus Einsen und Nullen aufgebaut ist. Wir wählen dafür eine Liste namens *tape*, da sich diese einfach in der Länge ändern lässt. Für die Darstellung erzeugen wir einige Bilder mit Einsen und Nullen unterschiedlicher Größe, wobei die gelben Versionen zur Anzeige der Kopfposition (*pos*) benutzt werden. Die Arbeitsgeschwindigkeit und die Zellengröße (*cell type*) sollen am Bildschirm veränderbar sein. Insgesamt benötigen wir damit die Variablen *tape*, *max tape length*, *pos*, *cell type* und *pause(ms)*.

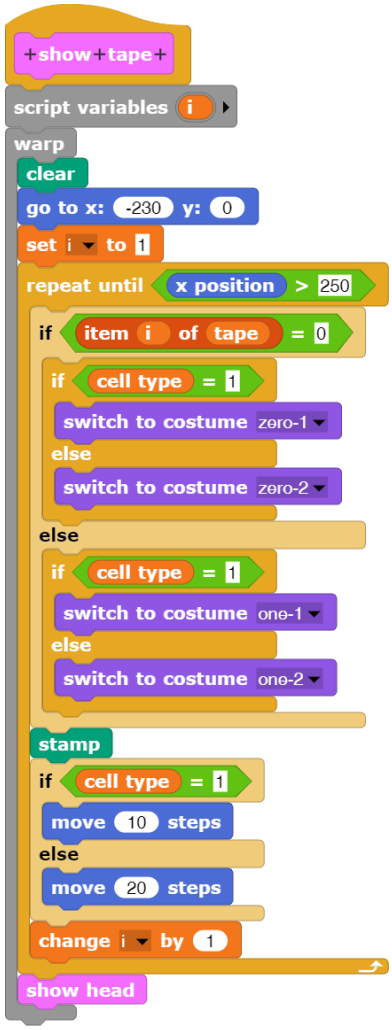
Die Anfangsbeschriftung muss erfragt und daraufhin ein entsprechendes Band erzeugt und angezeigt werden. Das erledigen wir durch die zeichenweise Aufspaltung des Eingabestrings in eine Liste. Vorne hängen wir die geforderten zwei Nullen an, und wir füllen das Band bei Bedarf mit Nullen bis zur Maximallänge auf.

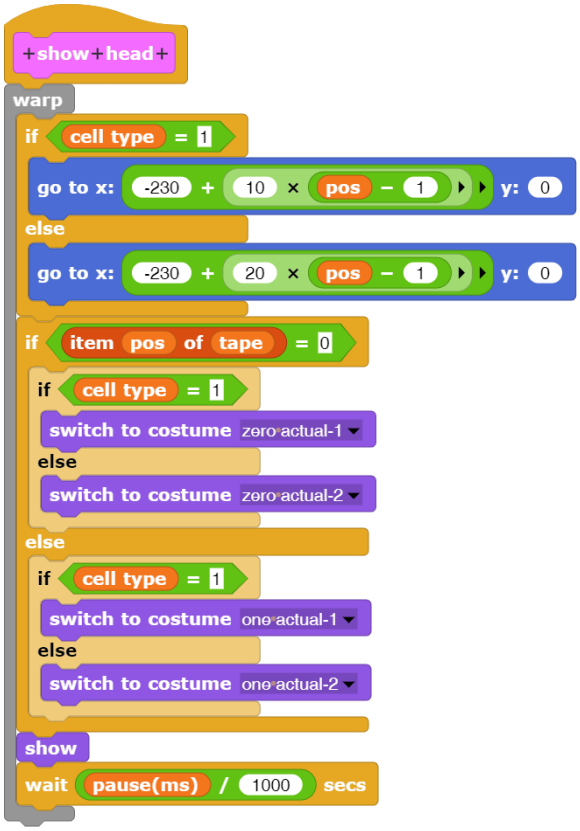


Auf diesem Band muss die Standardlage eingenommen werden, dabei wird der Wert der Variablen *pos* bestimmt, die die Lage des Kopfes angibt. Wie suchen, von rechts beginnend, die erste Eins.

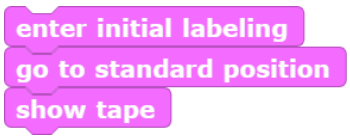


Danach wird das Band angezeigt, indem Bilder der Kostüme nebeneinander auf die Bühne „gestempelt“ werden. Zur Anzeige der Kopfposition berechnen wir dessen Bildschirmkoordinaten und wechseln in eins der gelben Kostüme.

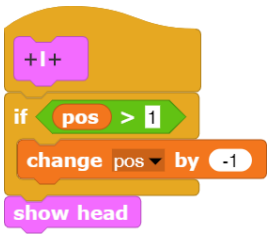


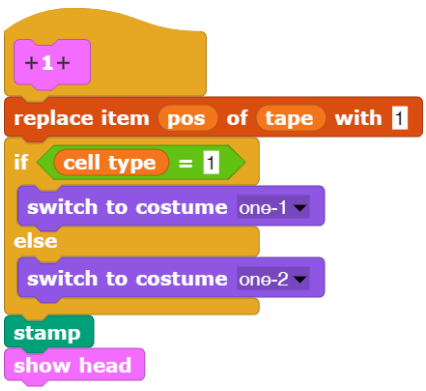


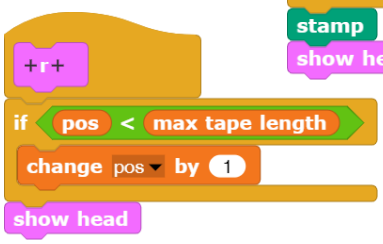
Insgesamt erhalten wir als Start-Befehlsfolge:



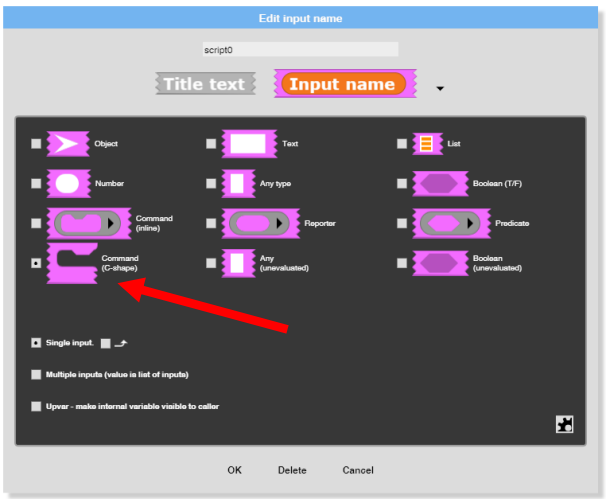
Die elementaren Maschinen lassen sich jetzt schnell erzeugen:



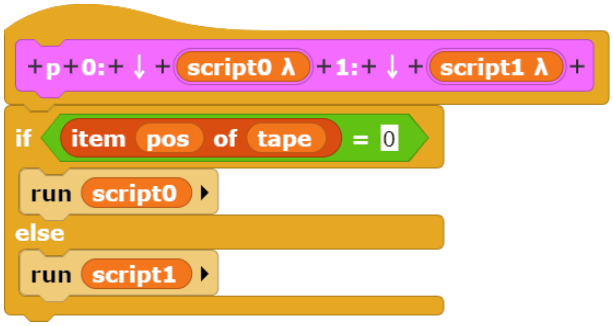




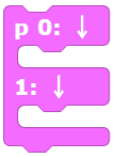
Etwas komplizierter ist die Erzeugung der Prüfmaschine *p*, denn diese muss ja zwei unterschiedliche Skripte ausführen können – je nach Bandbeschriftung. Diese Skripte dürfen also nicht VOR dem Aufruf der Maschine als Parameterwerte evaluiert werden, sondern es werden zwei Skripte übergeben, die NACH dem Aufruf je nach Bandbeschriftung auszuführen sind. Die „Parameterwerte“ sind also Skripte. Bei der Typisierung der Parameter wählen wir *Command (C-shape)* aus, um die Evaluation zu verhindern. Die Parameter werden durch ein λ als Skripte gekennzeichnet.



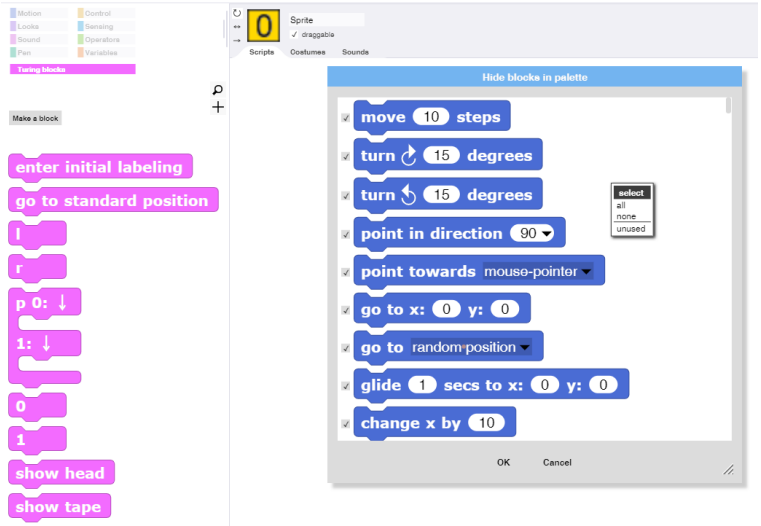
Die nach unten gerichteten Pfeile nach der Null oder der Eins findet man in der Auswahlliste, die sich hinter dem kleinen Pfeil hinter dem Namen von „*Title text*“-Parametern zeigt.



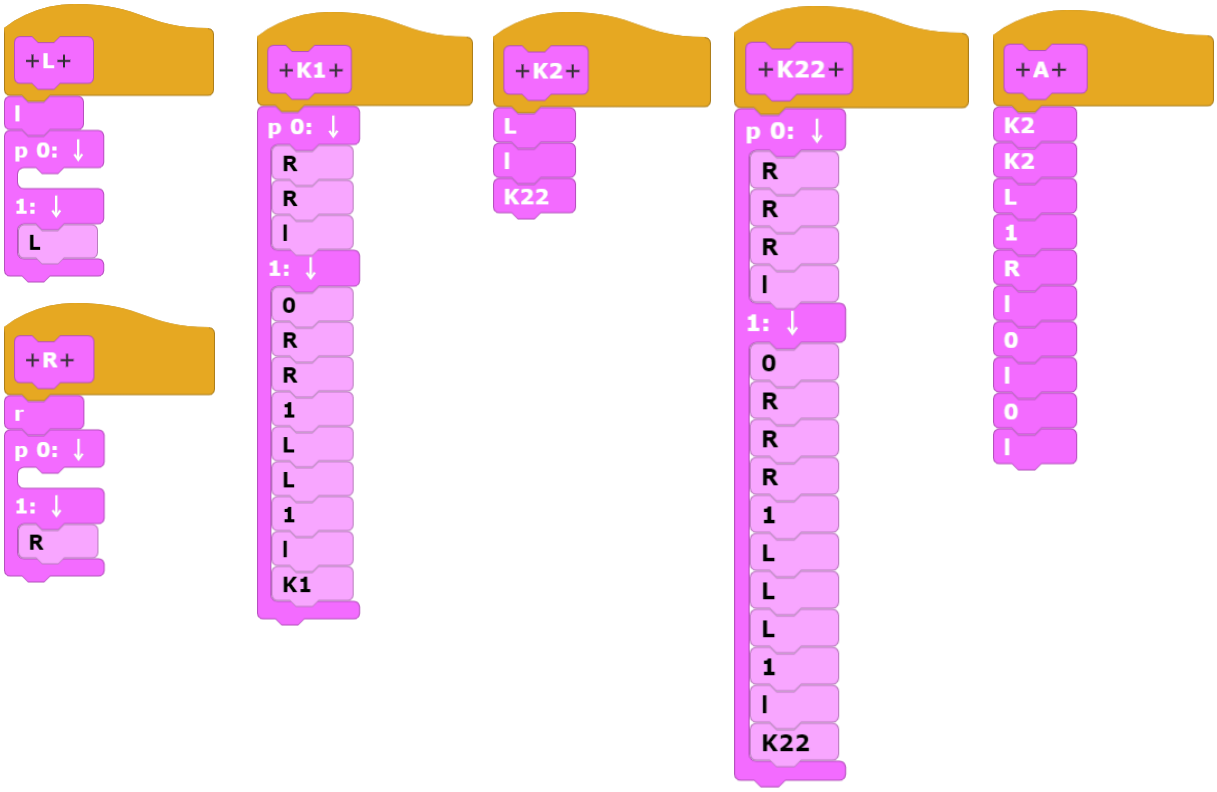
Der neue Block, eine Kontrollstruktur, hat dann das folgende Aussehen:



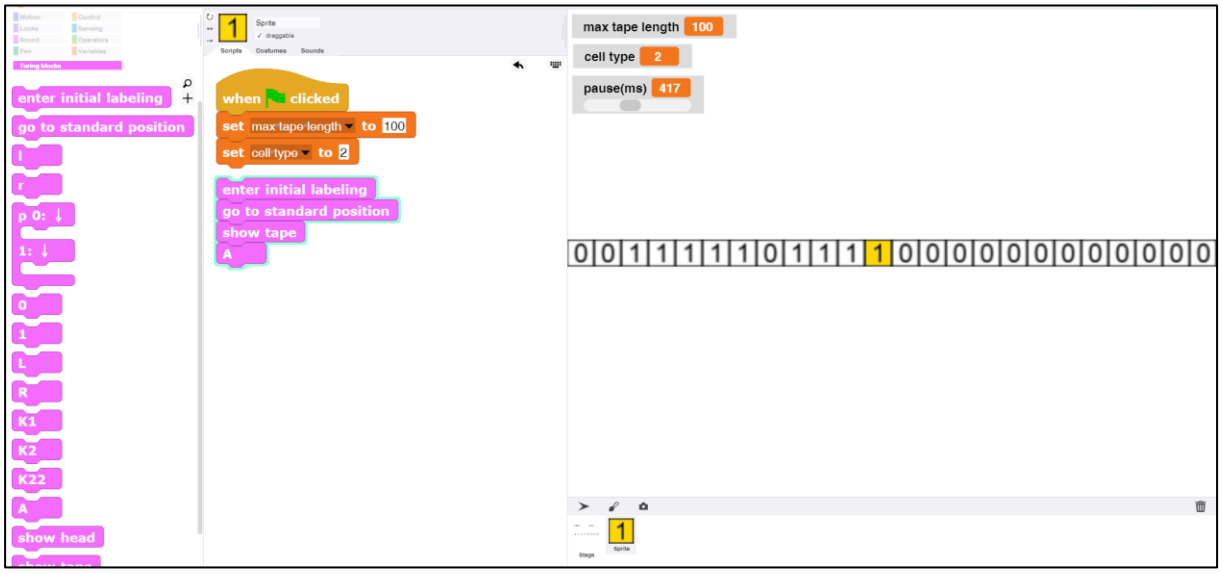
Wir wollen jetzt unsere gekoppelten Turingmaschinen aus diesen neuen Blöcken aufbauen. Um nicht in Versuchung zu geraten, die Standardblöcke mitzubenutzen, wählen wir nach Rechtsklick auf eine Palette (*hide blocks…*) und wählen alle Blöcke außer unseren neuen aus. Die Standardpaletten sind dann leer.



Mit diesen Maschinen lassen sich die anderen „ganz normal rekursiv“ in *Snap!* als Blöcke entwickeln.



Die Arbeit der Maschinen kann am Bildschirm in unterschiedlichen Geschwindigkeiten verfolgt und somit überprüft werden. Danach werden sie als neue Blöcke für komplexere Probleme benutzt.



16.4 Zelluläre Automaten: iteriertes Gefangenendilemma[69]

Altersstufe: *Sekundarstufe II* Material: *Cellular automaton*

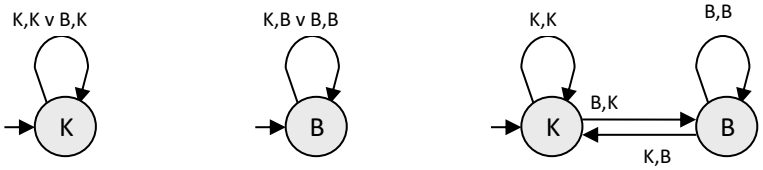
Wir wollen einen zellulären Automaten bauen, der auf dem Gefangenendilemma[70] aufbaut, aber etwas abgewandelt für den Handel im Internet. Das Verhalten der Handelspartner wird durch Automaten simuliert, die auf einem in beiden Dimensionen abgeschlossenen Gitter sitzen und innerhalb einer Von-Neumann-Nachbarschaft[71] Handel mit den Partnern treiben. Sie tauschen – wie im Internet üblich – Waren gegen Geld. Dabei gibt es unterschiedliche Arten von Geschäftspartnern:

- *Naive* kooperieren immer, liefern also den korrekten Gegenwert.
- *Betrüger* kooperieren nie.
- *Gewitzte* kooperieren anfangs und reagieren danach so, wie der Partner beim letzten Mal.

Wir beschreiben das Verhalten der Handelspartner durch Zustandsdiagramme:

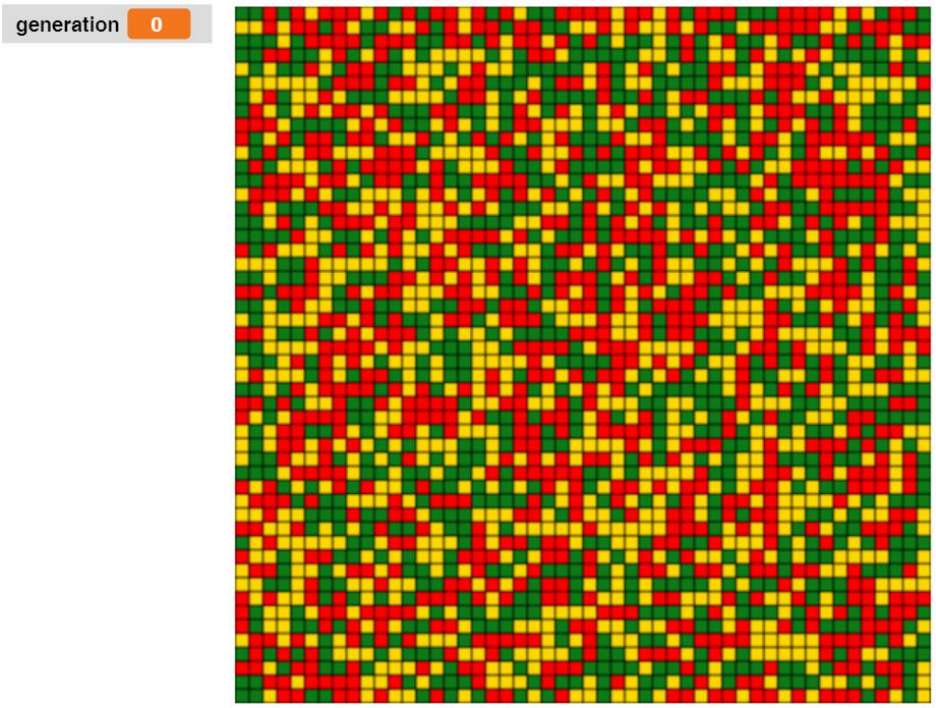
K: „kooperieren“

B: „betrügen“



Der „Naive“ Der „Betrüger“ Der „Gewitzte“

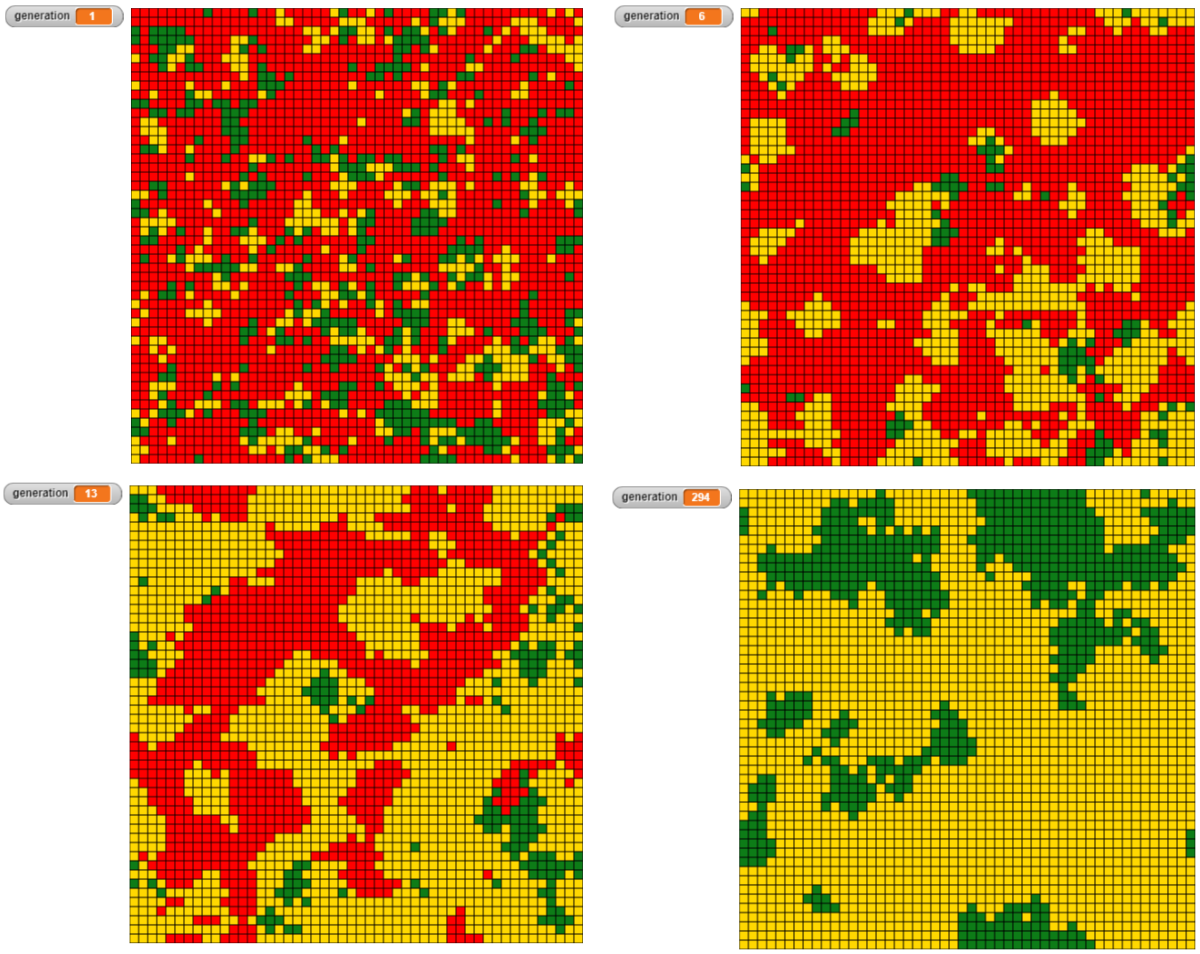
Ordnen wir solche Automaten in einem Gitter an, verteilen sie zufällig und färben sie entsprechend ihrem Zustand (grün als „Naiver“, rot als „Betrüger“ oder gelb als „Gewitzter“) ein, dann erhalten wir ein Bild ähnlich dem folgenden:



[69] nach Eckart Modrow, Zelluläre Automaten, LOG IN 127 (2004)
[70] https://de.wikipedia.org/wiki/Gefangenendilemma
[71] https://de.wikipedia.org/wiki/Von-Neumann-Nachbarschaft

Der weitere Ablauf ist einfach: Zuerst handeln alle Partner einmal mit ihren Nachbarn aus der Von-Neumann-Nachbarschaft, also mit den Nachbarn oben, unten, links und rechts. Danach bewerten alle Partner den Erfolg ihrer Nachbarn. Als Opportunisten übernehmen sie den Zustand des erfolgreichsten Nachbarn oder behalten ihren Zustand bei, wenn sie selbst besser waren.

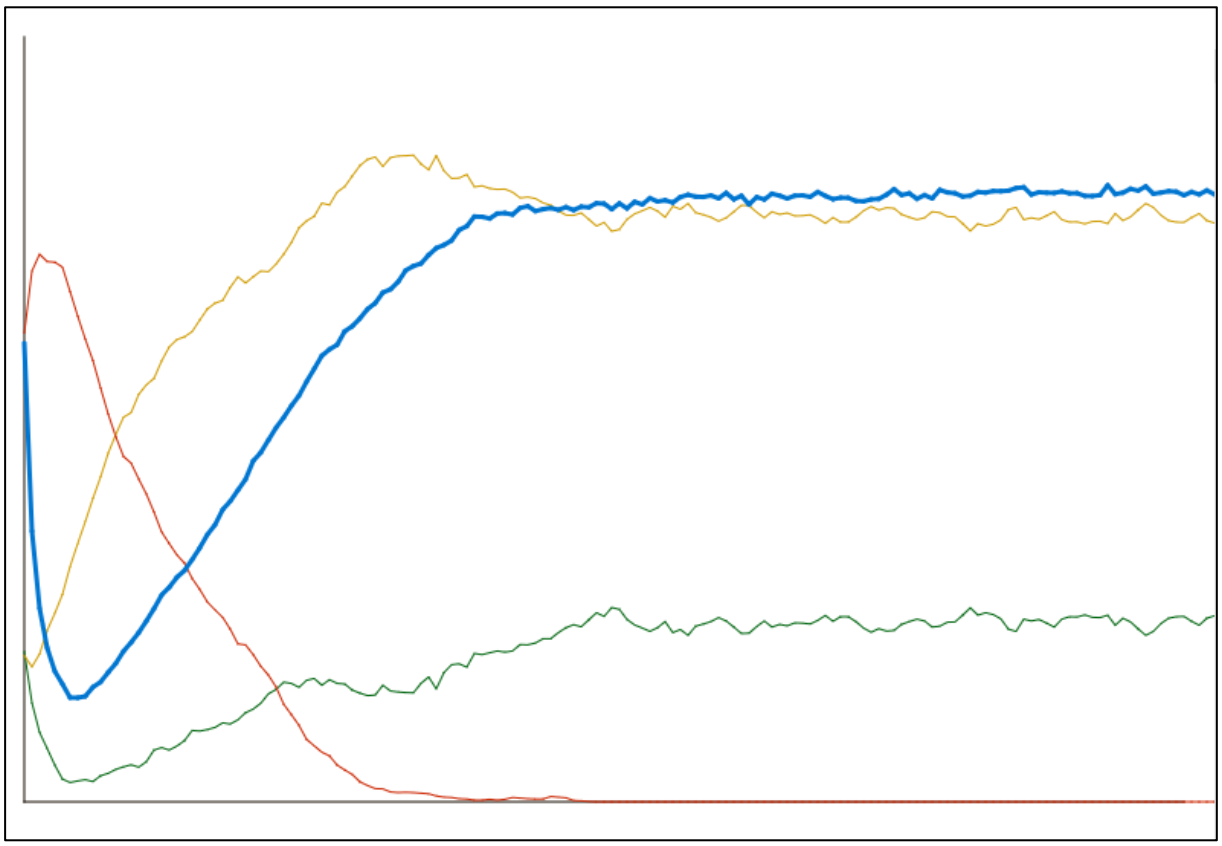
In den ersten Generationen setzen sich meist die „Betrüger“ durch. Doch danach bilden sich Cluster aus „Naiven“ bzw. „Gewitzten“, und dann beginnt eine wilde „Schlacht“.



Zwar werden die „Naiven“ hart von den „Betrügern“ bedrängt. Sie halten sich aber ganz gut in Gruppen. Die „Gewitzten“ setzen sich gegenüber den „Betrügern“ – je nach Konfiguration – meist durch und kooperieren mit den „Naiven“. Am Ende siegen meist die „Gewitzten“ – aber eben nicht immer. In Gruppen betrügen sich die „Betrüger“ gegenseitig und gewinnen so nichts, während die „Gewitzten“ sich gegen sie behaupten und mit den Naiven „im Rücken“ erfolgreicher sind. Die Abläufe hängen stark davon ab, wie das unterschiedliche Verhalten gewichtet wird.

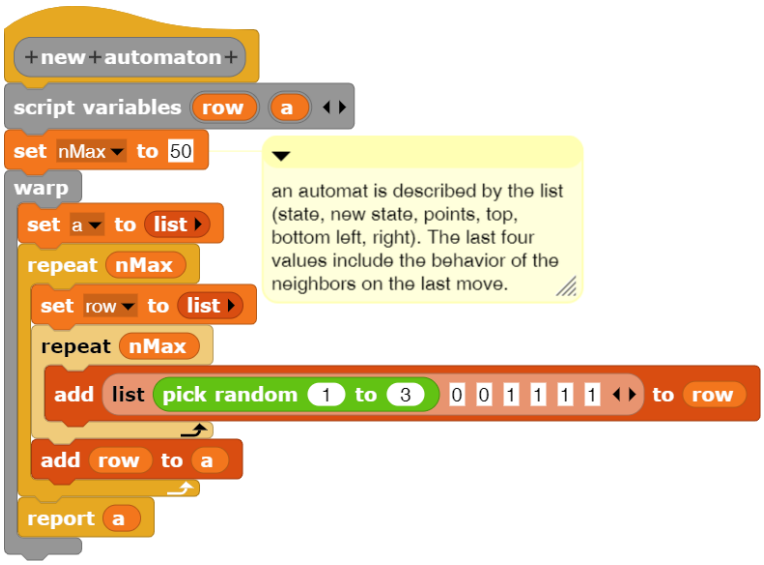
Zur Bewertung des Systems sind globale Größen geeignet, z. B. ein „Bruttosozialprodukt“ als Summe aller Handelspunkte. Die Beobachtung der manchmal überraschenden Abläufe liefert Ansatzpunkte zur Diskussion ethischer Fragen. Auch wenn das Beispiel natürlich nicht direkt auf gesellschaftliche Systeme übertragbar ist, so haben wir doch ein für die meisten neuartiges Argument für kooperatives, soziales Verhalten gefunden, das nicht aus transzendenten oder philosophischen Überlegungen gewonnen wird, sondern aus Effizienzbetrachtungen. Es steht darin in klarem Gegensatz zur Egozentrik des Primitiv-Darwinismus, die oft die öffentliche Diskussion in dieser Hinsicht beherrscht. Als Beispiel mag ein Diagramm dienen, in dem einerseits die Gesamtzahlen der drei Automatentypen (Naiver, Betrüger, Gewitzter) aufgetragen wurden, und dazu etwas dicker in Blau die Summe der insgesamt von allen Typen erreichten Handelspunkte, also

das „Bruttosozialprodukt“. Man sieht sehr schön, dass der „gesellschaftliche Wohlstand“ (wenn man den aus dem „Handelsvolumen“ ableiten will) gegenläufig zur Anzahl der „Egoisten“ ist – natürlich bei den eingestellten Bedingungen. Unter diesen sterben Betrüger mangels Erfolgs meist aus, und die Naiven harmonieren mit den Gewitzten prächtig – wenn sie denn unter sich sind. Wird das Verhalten anders gewichtet, dann können Betrüger durchaus erfolgreich sein. Es hängt also von den Spielregeln ab, wer Erfolg hat. Man sollte über diese nachdenken, nicht nur in einer Simulation!



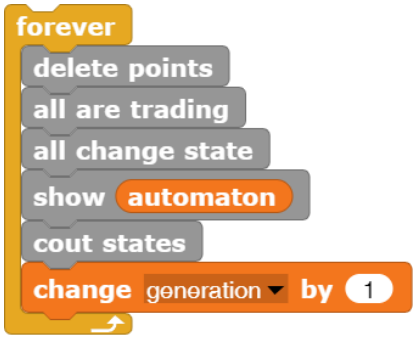
Programmtechnisch ist das System eher einfach, allerdings durch die Wechsel der Blickrichtung manchmal umfangreich.

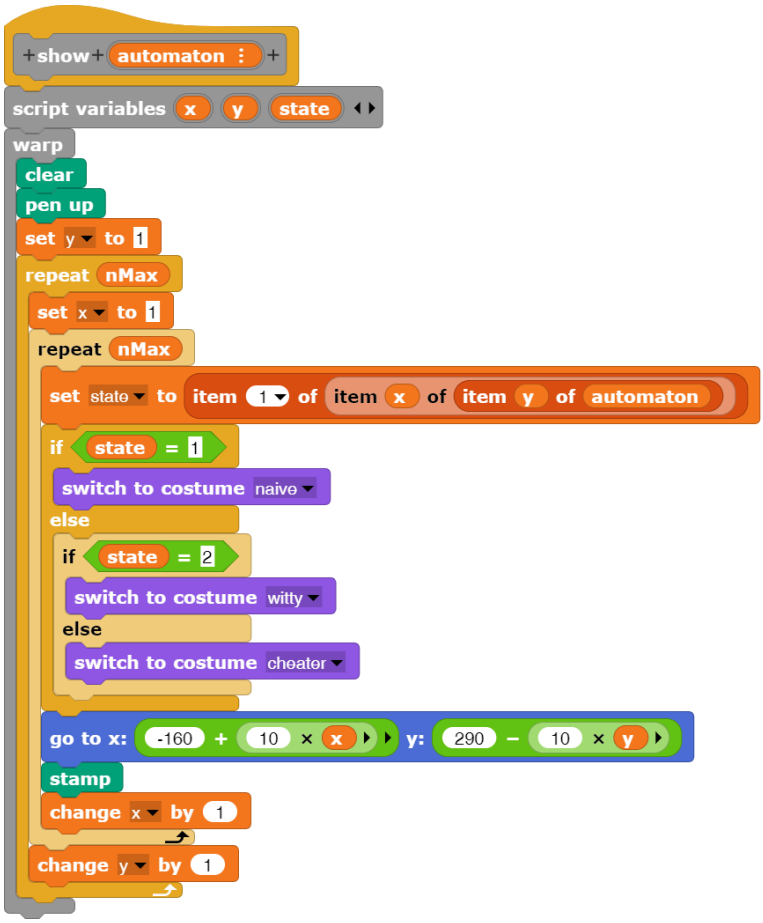
Ein neuer Automat kann durch eine Liste von Listen beschrieben werden, wobei die Automaten an den Gitterplätzen Zahlenfolgen entsprechen, die einerseits ihren Zustand und die erreichten Handelspunkte, andererseits das „Gedächtnis“ über das vergangene Verhalten der Nachbarn enthalten.



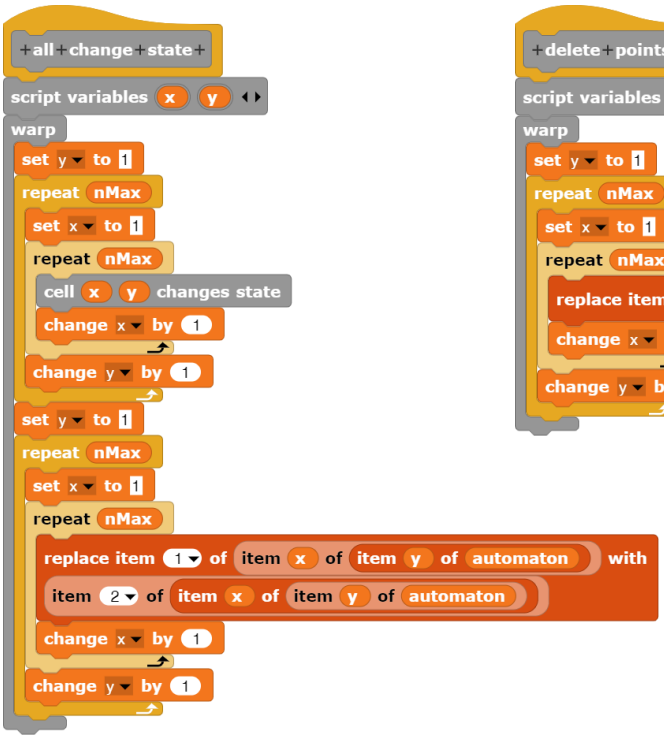
Der Gitterautomat kann dargestellt werden, indem verschiedenfarbige Kostüme (kleine Rechtecke) nebeneinander auf den Arbeitsbereich gestempelt werden. Der ist vorher auf die Größe 800x600 Pixel geändert worden.

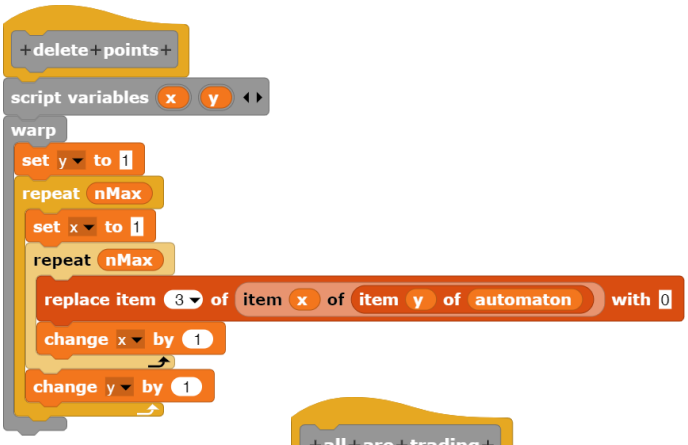
Ist der Automat einmal erzeugt, dann werden die neuen Generationen aus der jeweils letzten erzeugt.





Die Skripte sind sehr ähnlich aufgebaut: es wird jeweils über alle Gitterplätze iteriert.





Der Handel einer Zelle mit den Nachbarn hängt einerseits von den Zuständen der Teilautomaten, andererseits von deren bisherigem Verhalten ab. Da diese Daten in den Automatenwerten gespeichert wurden, lassen sie sich leicht abfragen. Dargestellt ist der Handel mit dem linken Nachbarn:

Zelle bestimmen

Toruswelt: die gegenüberliegenden Ränder sind miteinander verbunden.

Nachbarzelle bestimmen

kooperiert die Zelle?

kooperiert der Nachbar?

Verhalten des Nachbarn „für später“ speichern

Wenn beide kooperieren: Gewinn zwischen 2 und 10, sonst nix

Der Nachbar wird betrogen: Gewinn zwischen 1 und 20

beide betrügen: fast kein Gewinn

Der Handel mit den anderen drei Nachbarn erfolgt entsprechend. Die Unterschiede liegen nur in den Positionen des gespeicherten Vorverhaltens.

Sind die Werte einer Generation ermittelt, dann lassen sie sich zählen und in einer Liste zusammenstellen – und aus dieser entsteht ein Diagramm.

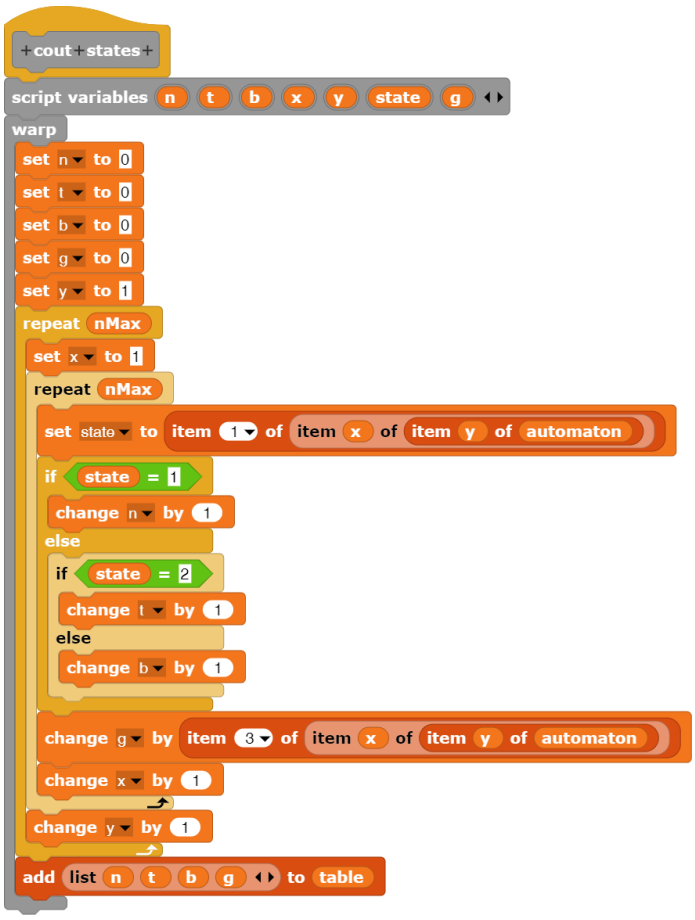
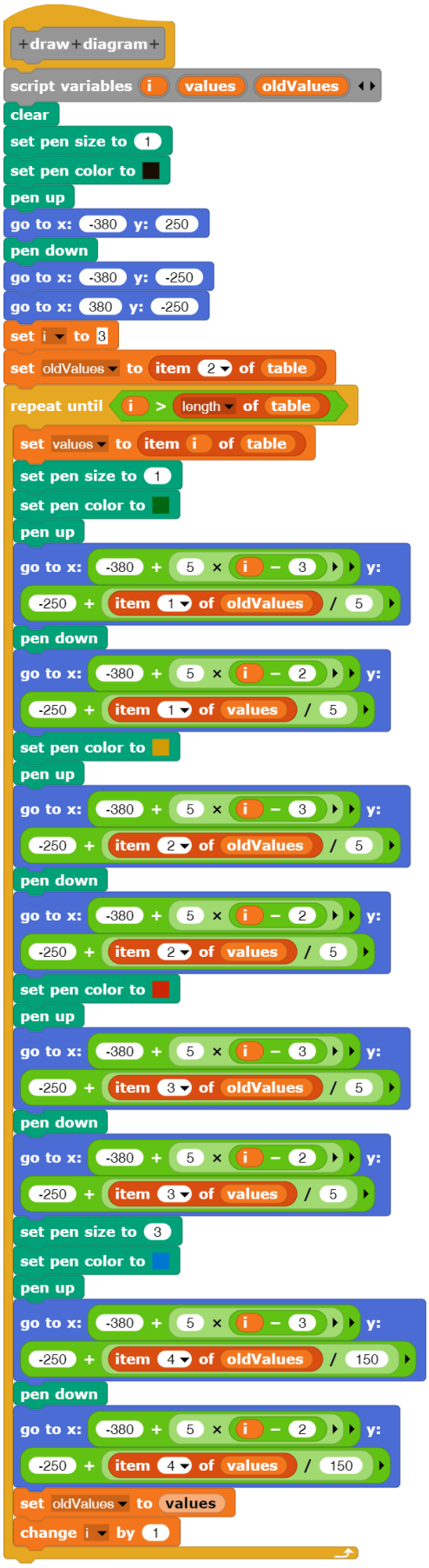


Table view

34	A	B	C	D
1	Naive	TitForTat	Cheater	overall
2	489	428	1583	45677
3	320	428	1752	26541
4	243	485	1772	18849
5	177	589	1734	16254
6	161	684	1655	14581
7	130	786	1584	14178
8	125	882	1493	13819
9	103	993	1404	14482
10	119	1133	1248	14636
11	118	1281	1101	16092
12	121	1394	985	17108
13	143	1478	879	18450
14	158	1577	765	19548
15	168	1673	659	20786
16	193	1741	566	22002
17	224	1756	520	23629
18	219	1790	491	24887
19	247	1792	461	26267
20	268	1796	436	27174

OK



16.5 Aufgaben

1. Entwickeln Sie einen endlichen Automaten als Prädikat zur Erkennung
   a: korrekter **KFZ-Kennzeichen** aus Hamburg, Bremen und Hannover.
   b: korrekter **IBAN-Nummern**. Sie können die Suche gerne auf einige wenige Banken beschränken.
   c: genügend komplexer **Passwords**. Definieren Sie vorher, was „genügend komplex“ bedeutet.

2. Verbessern Sie die **Silbentrennung** durch die Berücksichtigung
   a: von Doppelkonsonanten.
   b: von typischen Vorsilben.

3. Entwickeln und testen Sie eine **gekoppelte Turingmaschine**,
   a: die eine Einsengruppe über eine andere hinweg kopiert (*K2*).
   b: die eine Einsengruppe nach links an eine andere soweit heranschiebt, bis die beiden Gruppen nur noch durch eine Null getrennt sind.
   c: die zwei natürliche Zahlen miteinander multipliziert.
   d: die eine 1 hinter zwei Einsengruppen schreibt, wenn diese gleichlang sind, sonst eine Null.
   e: die zwei natürliche Zahlen subtrahiert – wenn das möglich ist. Wenn nicht, dann dreht sie durch: sie läuft nach rechts weg.

4. a: Ersetzen Sie beim betrachteten **Gitterautomaten** den Handel aller Teilautomaten mit den Nachbarn „pro Runde“ durch einen zufallsgesteuerten Prozess, in dem Automaten mit benachbarten (mit beliebigen) Partnern handeln.
   b: Ersetzen Sie die Von-Neumann- durch eine **Moore-Nachbarschaft**.
   c: Der Automat kann leicht zu einem Ising-Modell umgebaut werden, indem die Automaten als **Spingitter** betrachtet werden. Pro Runde kippt die Mehrheit der benachbarten Spins den Spin in der Mitte in ihre Richtung. Es ergeben sich unterschiedlich magnetisierte Bereiche.

5. a: Informieren Sie sich über **lineare zelluläre Automaten** nach Stephen Wolfram.
   b: Realisieren Sie das Modell.

17 Projekte

17.1 LOGO für Arme

Altersstufe: *Sekundarstufe I/II* Material: *LOGO for the poor*

Wir wollen eine kleine Programmiersprache entwickeln, mit deren Hilfe wir Programme für eine Turtle schreiben können – also für jedes *Snap!*-Sprite. Das Projekt soll zeigen, wie eine *textbasierte Sprache* arbeitet und wie die Fehlermeldungen zustande kommen. Wir reduzieren das Problem etwas dadurch, dass wir nur Ein-Buchstaben-Befehle zulassen. Betrachten wir die Möglichkeiten des in *Snap!* verwendeten Pens und wählen einige davon aus, dann erhalten wir einen möglichen Befehlssatz (hier sehr klein):

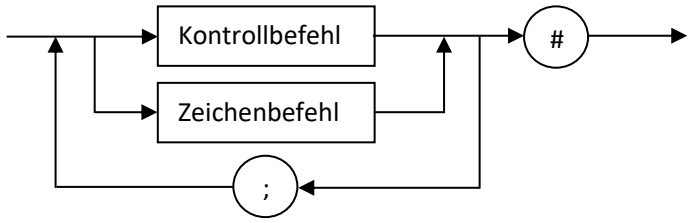
Mn bewegt die Turtle um die Strecke der Länge *n* in der augenblicklichen Richtung
Tn dreht die Turtle auf der Stelle um *n* Grad
U hebt den Stift
D senkt den Stift ab

Diese vier Befehle ergänzen wir um eine Kontrollstruktur, hier: eine Schleife – und fertig ist die Minimalversion einer Programmiersprache.

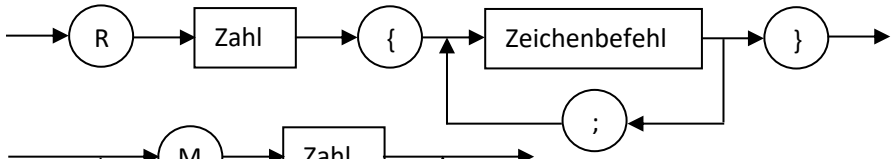
Rn{Zeichenbefehle}

Diese Rohskizze gießen wir in Form von Syntaxdiagrammen: Ein Turtleprogramm besteht aus einer Folge von Befehlen, die durch Semikolons getrennt und von einem Doppelkreuz-Zeichen abgeschlossen werden.

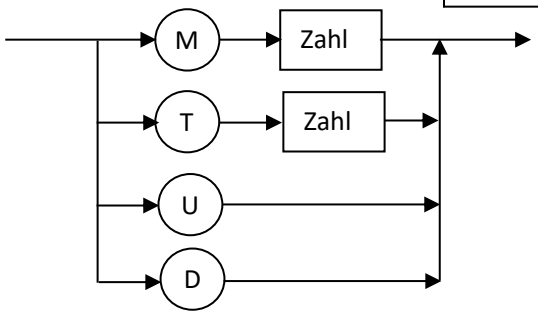
Turtleprogramm:



Kontrollbefehl:



Zeichenbefehl:

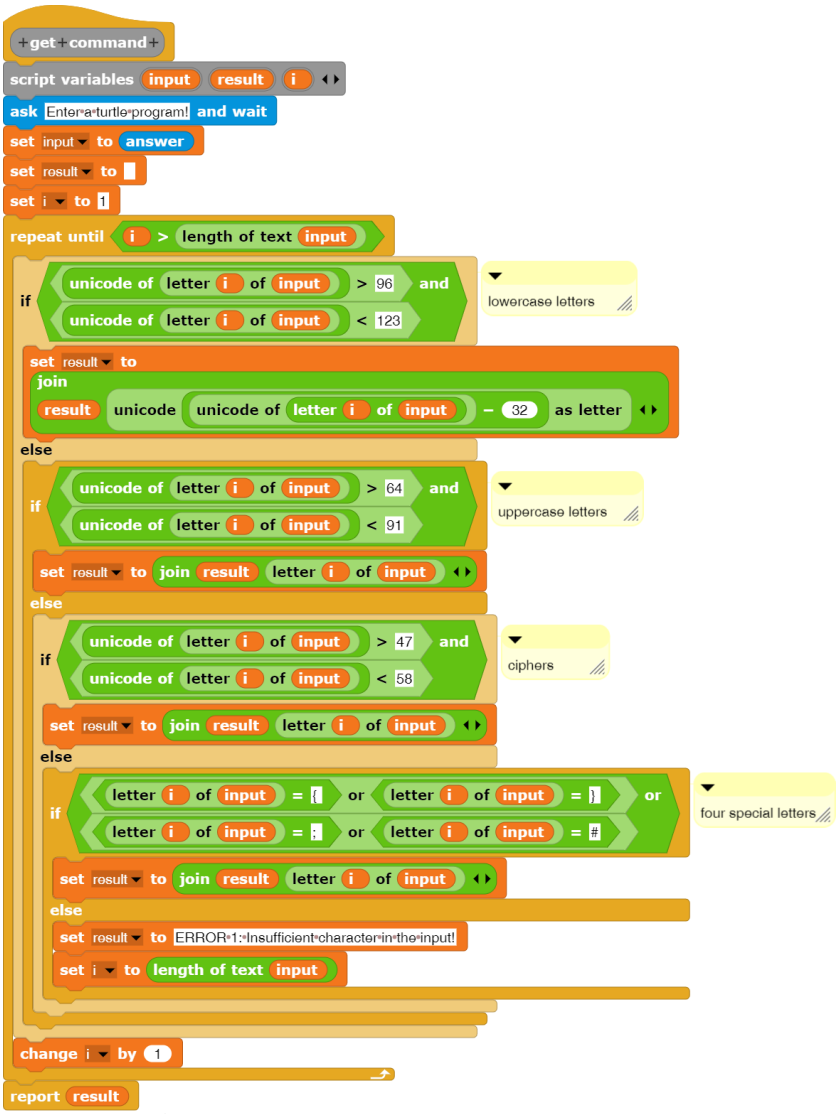


Zahlen: natürliche Zahlen

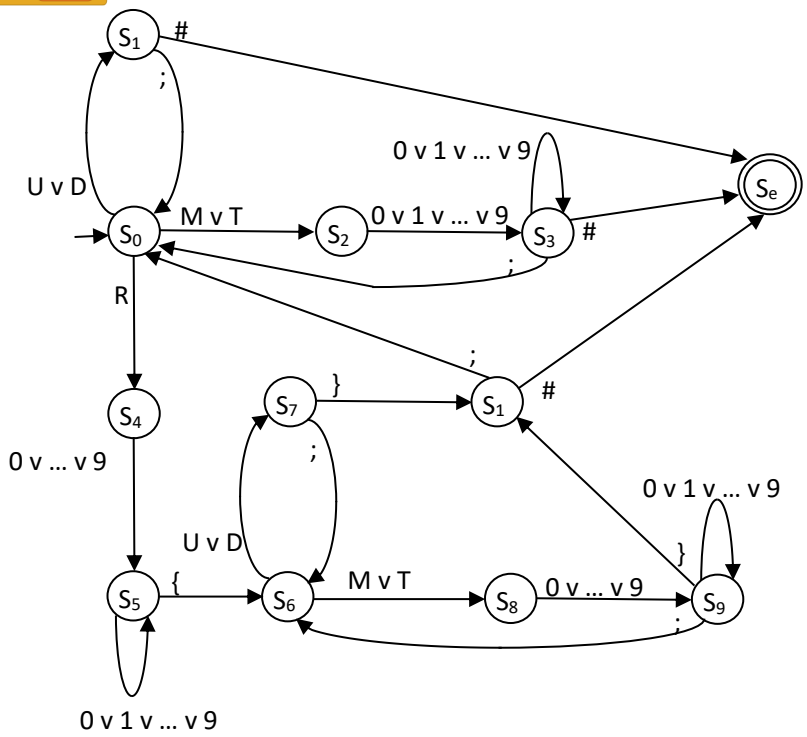
Programme sind also z. B.: *D;R4{M100;T90};U#*
M100;T90;M100;T90;M100;T90;M100;T90#
D;R180{M200;T183};R360{M1;T1}#

Wir gehen davon aus, dass überflüssige Zeichen wie Leerzeichen zuerst aus dem Programm entfernt werden. Das können wir z. B. erreichen, indem wir eingegebene Kleinbuchstaben in Großbuchstaben verwandeln sowie Ziffern und die vier Sonderzeichen „;“, „#“, „{„ und „}“ zulassen. Alle anderen Zeichen führen zur Fehlermeldung *„ERROR 1: unzulässiges Zeichen in der Eingabe!*“

Wir schreiben also eine einfache Eingabemethode mit Zeichenkontrolle.



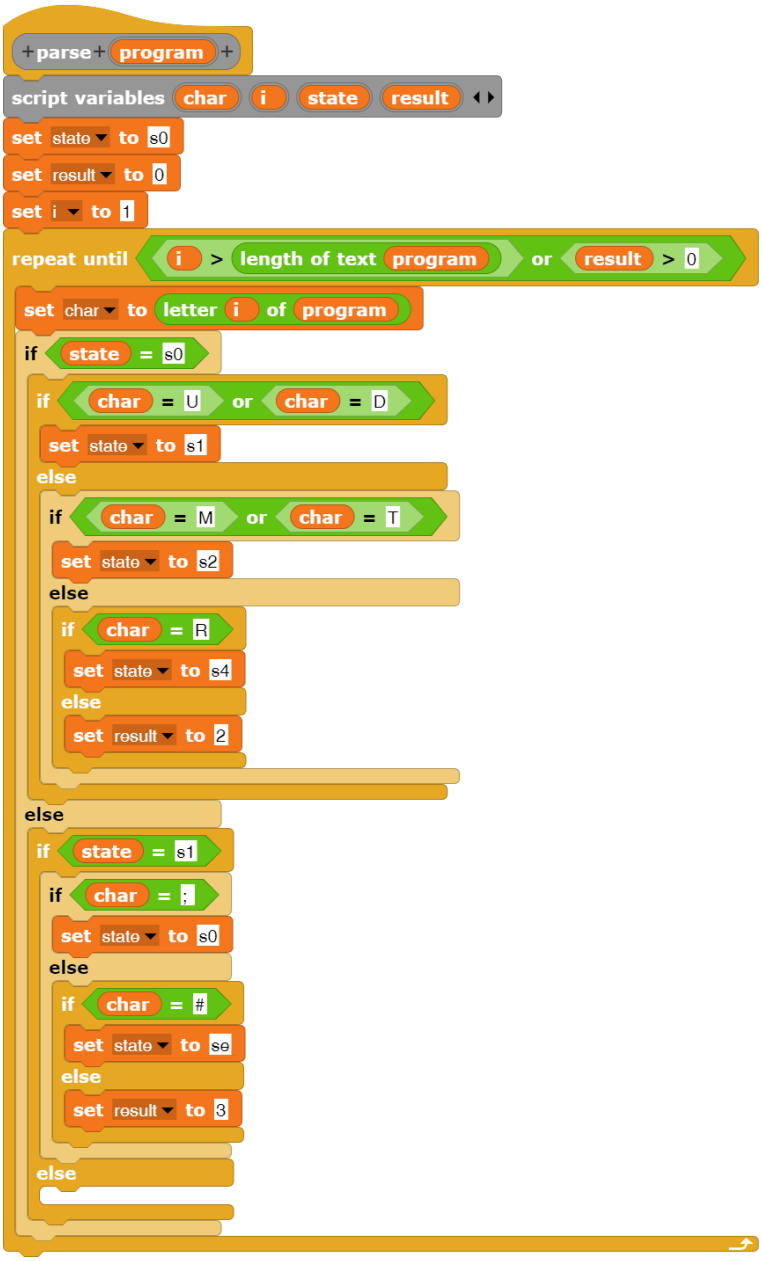
Die Eingabe muss darauf kontrolliert werden, ob sie ein zulässiges LOGO-Programm darstellt – sie wir „geparst“. Den Parser können wir in diesem Fall als endlichen Automaten darstellen[72]. Die nicht angegebenen Übergänge führen in einen Fehlerzustand.



[72] Weshalb eigentlich?

In den einzelnen Zuständen können wir entscheiden, welche Zeichen in Folgezustände führen, und welche nicht. Damit können wir bei Fehleingaben auch angeben, welche Zeichen eigentlich erwartet wurden. Nummerieren wir diese Fehlermeldungen des Parsers in der Reihenfolge ihres Auftretens durch, dann erhalten wir die nebenstehende Tabelle. Wenn wir die Position des Zeichens im Befehl, an der der Fehler auftrat, mit auswerten, dann können wir den Fehler sogar anzeigen.

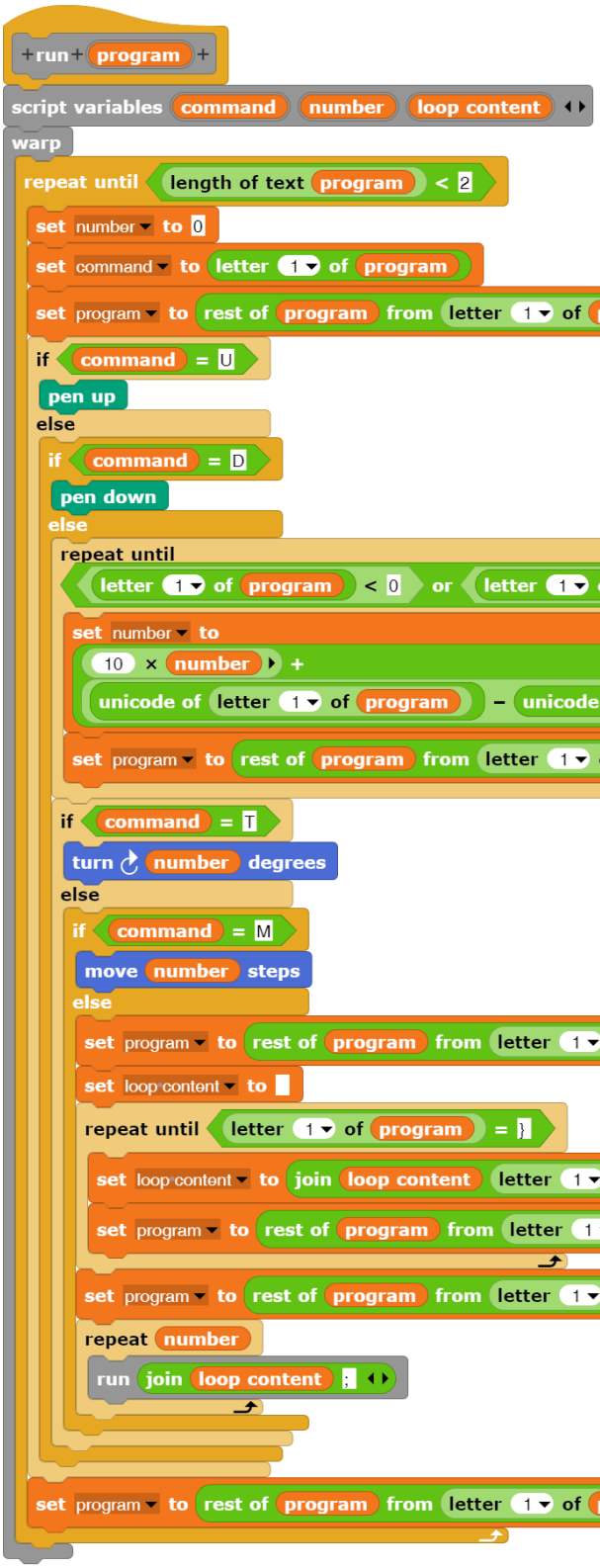
Zustand	mögliche Fehlermeldung
S_0, S_6	2: unbekannter Befehl
S_1, S_{10}	3: <;> oder <#> erwartet
S_2, S_4, S_8	4: Zahl erwartet
S_3	5: Zahl, <;> oder <#> erwartet
S_5	6: Zahl oder <{> erwartet
S_7	7: <;> oder <}> erwartet
S_9	8: Zahl, <;> oder <}> erwartet
	9: unerwartetes Ende der Eingabe



Die Übersetzung des Parsers besteht nur aus einer ellenlangen Abschrift des Zustandsgraphen – aus lauter geschachtelten Alternativen. Wir zeigen hier nur den ersten Teil.

Der Parser *parse <program>* wird durch die Zeichenfolge des Programms durch das Zustandsdiagramm geführt. Gibt es in einem Zustand keinen zulässigen Übergang, dann meldet er den entsprechenden Fehler durch den Wert der Variablen *„result“*. Korrekte Programme haben als Ergebnis den Wert 0.

Der Interpreter *run <program>* kann davon ausgehen, dass das eingegebene Programm fehlerfrei ist – schließlich wurde es ja geparst. Deshalb kann er sich nacheinander jeweils das erste Zeichen des Programms merken – das ist der nächste Befehl – und dieses Zeichen löschen. Je nach Befehl führt er dieses aus und sucht sich ggf. die erforderlichen Parameter zusammen, z. B. den Drehwinkel. Alle verarbeiteten Zeichen werden gelöscht. Damit endet die Bearbeitung, wenn das Programm nur noch aus dem letzten Zeichen – dem „#“ – besteht.



Das Programm wird zeichenweise abgearbeitet, die verarbeiteten Zeichen werden gelöscht. Dafür benutzten wir die Funktion *rest of* … unserer String-Bibliothek.

PenUp-Befehl (U)

PenDown-Befehl (D)

Zahl zusammensuchen

Turn-Befehl (T)

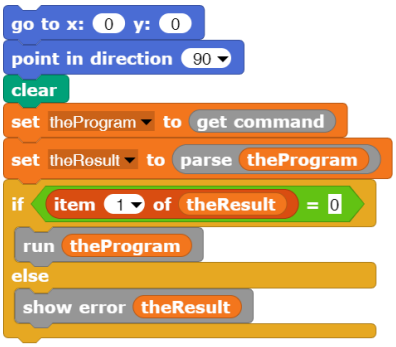
Move-Befehl (M)

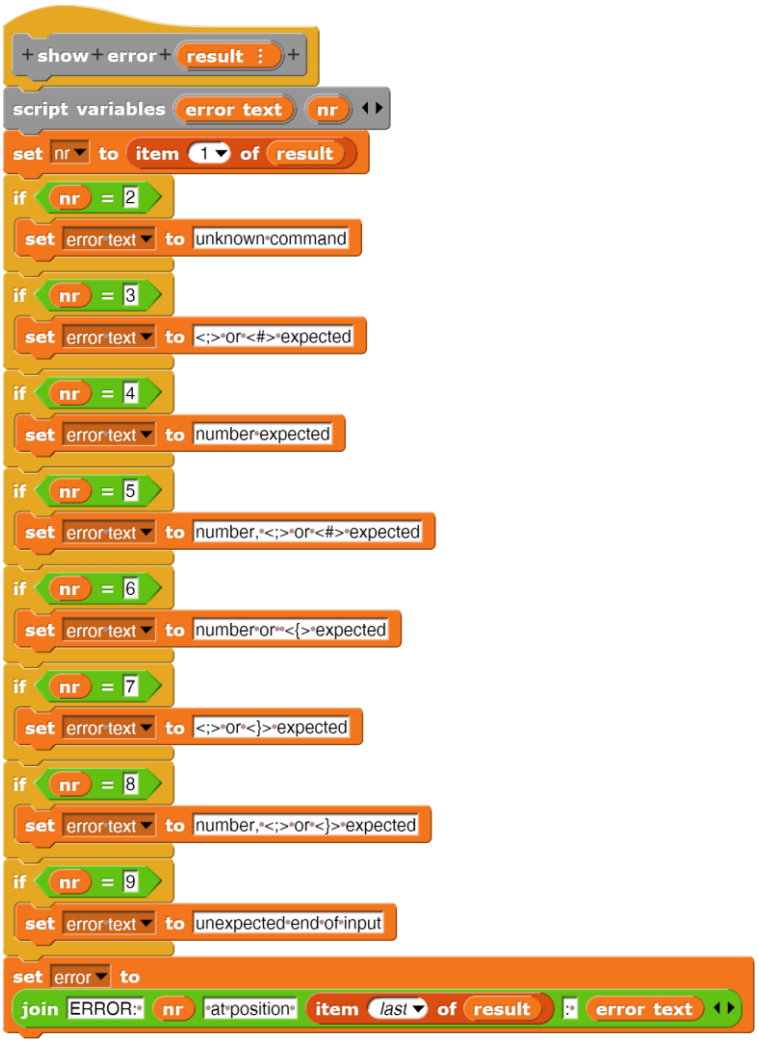
Schleife (R) ausführen. Dazu Schleifeninhalt bis zum nächsten „}“ zusammensuchen …

… und so oft ausführen, wie die Zahl angibt. Dabei ein „;“ an den Schleifeninhalt anhängen.

Wenn wir noch die Fehlermeldungen im Klartext ausgeben, dann wird unsere Programmiersprache langsam brauchbar.

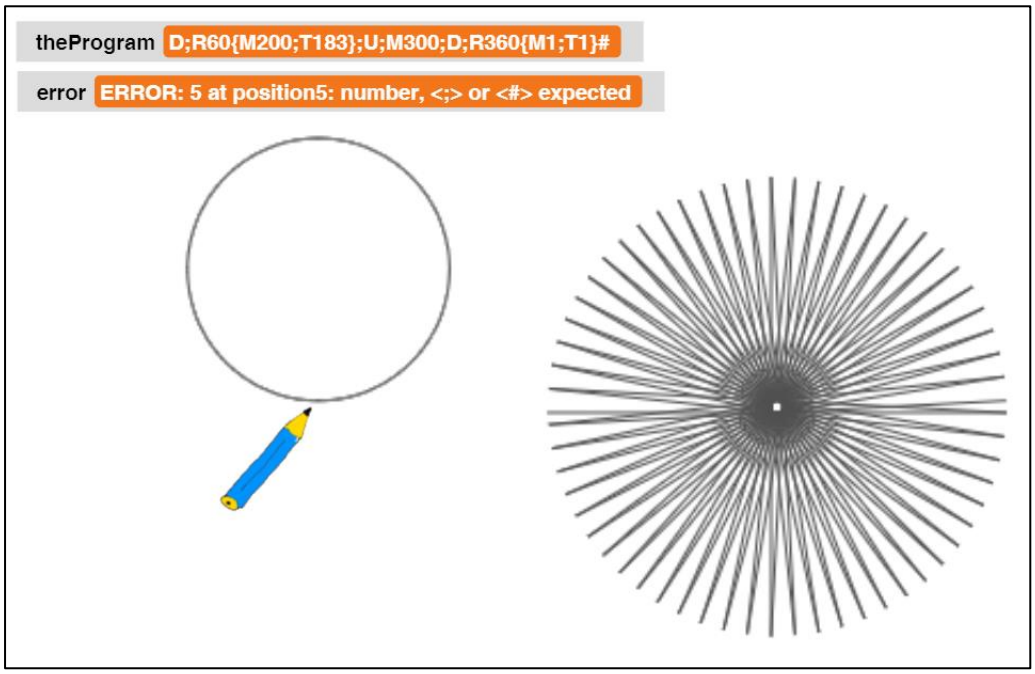
Wir können Programme durch ein kurzes Skript auswerten.





Wir sollten uns klarmachen, dass die Definition dieser Sprache rein willkürlich ist. Der Rumpf der Schleife könnte statt mit geschweiften Klammern auch mit eckigen, mit Prozentzeichen oder Smileys eingeschlossen werden, und dass Anweisungen durch Semikolons getrennt, aber nicht beendet werden, entspringt ebenfalls nur der aktuellen Laune. Ein Programm ist syntaktisch „richtig“, wenn es der Sprachdefinition entspricht, und die wiederum entspricht den Vorstellungen der die Sprache Entwickelnden. Sie folgt nicht aus allgemeingültigen Regeln.

Auch aus den Fehlermeldungen lässt sich lernen. Sie geben an, wo ein Fehler bemerkt wird, nicht, wann er gemacht wurde. Die angegebene Fehlerposition kann also weit hinter dem tatsächlichen Ort des Fehlers liegen.



Eigentlich ist es ja etwas seltsam, in einer grafischen Programmiersprache eine sehr primitive textbasierte Sprache zu entwickeln. Die Erfahrung zeigt aber, dass Lernende meist die Arbeit von Informatiker/innen mit der Entwicklung kryptischer Texte verbinden – sprich: auch mal „richtig“ programmieren wollen. Wir können diesem Wunsch entgegenkommen, wenn wir solch eine Minisprache auf einem Standardgebiet der Informatik, in diesem Fall der Automatentheorie, einsetzen. Da wir sie selbst entwickeln, fördern wir nebenbei das Verständnis für die Verarbeitung von Texten, die ja auf vielen Ebenen in Informatiksystemen erfolgt. Außerdem haben wir ein stark differenzierendes, für arbeitsteiliges Vorgehen geeignetes und Aktivitäten herausforderndes Thema gefunden, das schnell zu vorzeigbaren Ergebnissen führt.

Aufgaben:

1. **Erweitern** Sie die Sprache LOGO für Arme durch
   a: einen Befehl Home (H), der die Turtle zur Bildschirmmitte schickt.
   b: einen Befehl Clear (C), der den Bildschirm löscht.
   c: einen Befehl Farbe<n> (Fn), der die Auswahl einer Stiftfarbe ermöglicht.
   d: einen Befehl TurnTo<winkel> (Nn), der die Turtle auf einen bestimmten Winkel dreht.
   e: einen Befehl MoveTo<x><Y> (Vx,y), der die Turtle zu einem bestimmten Punkt schickt.

2. Entwickeln Sie einen **Scanner**, der es gestattet, die Turtlebefehle in Langform einzugeben, also z. B. Turn 90 statt T90 zu schreiben. Der Scanner soll diese Befehle erkennen und sie in der Kurzform wieder ausgeben.

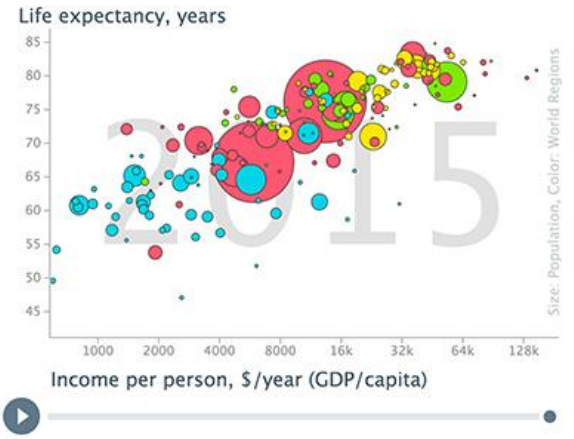
3. Führen Sie eine **Alternative** ein: In Abhängigkeit von der Farbe des Pixels am Ort der Turtle sollen unterschiedliche Zeichenbefehlsfolgen ausgeführt werden können. Reduzieren Sie die Syntax geeignet und implementieren Sie den Befehl.

4. Auf entsprechende Art sollen zwei **Schleifenarten** eingeführt werden: Die Turtle soll Zeichenbefehle ausführen, solange (WHILE) bzw. bis (DO) die Turtle sich über Pixeln einer anzugebenden Farbe befindet. Lassen Sie auch positionsabhängige Prädikate zu.

17.2 SnapMinder von Jens Mönig[73]

Altersstufe: *Sekundarstufe I/II* Material: *SnapMinder*

Das Programm beruht auf den Daten der Gapminder-Stiftung[74], die Tools zur Visualisierung statistischer Daten im Netz bereitstellt. Eins davon zeigt die Entwicklung der Länder in der jüngeren Vergangenheit, wobei die Lebenserwartung über dem Einkommen dargestellt wird und die Größe der „Blasen“ der Gesamtbevölkerung des Landes in einem Jahr entspricht. Bewegt man den Regler, dann kann man die zeitliche Entwicklung der Länder in diesem Koordinatensystem sehr eindrucksvoll verfolgen. Das Programm ist für mich ein wundervolles Beispiel, wie durch Visualisierung Auffälligkeiten in Daten erkannt werden *(„Weshalb fällt ein Land plötzlich nach unten?“ „Warum bewegt sich ein Land im Kreis?“, …)*, deren Ursachen dann erforscht werden können.



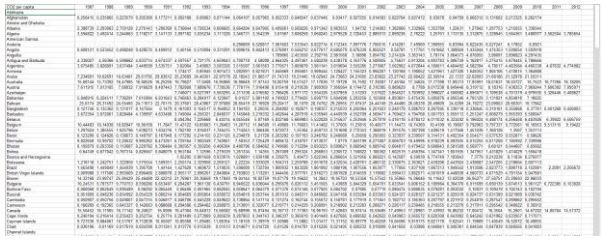
Die benutzten Daten – und viele andere – findet man unter https://www.gapminder.org/data/ in Tabellenform.

Import von Tabellendaten

Um die benötigten Daten zu importieren, laden wir die Datei in ein Tabellenkalkulationsprogramm und speichern sie gleich wieder als Tabulator-getrennte Textdatei. Als Beispiel nehmen wir den CO2-Ausstoß pro Person von 1751 bis 2012[75]. Für die ersten Jahre finden wir nur wenige Werte, aber dann wird es dicht.

Die erzeugte Textdatei lesen wir in eine Variable über deren Kontextmenü (*import…*) ein. Dazu muss sie im Arbeitsbereich angezeigt werden. Wir erhalten eine sehr lange Zeichenkette.

Diese verwandeln wir in eine Liste:

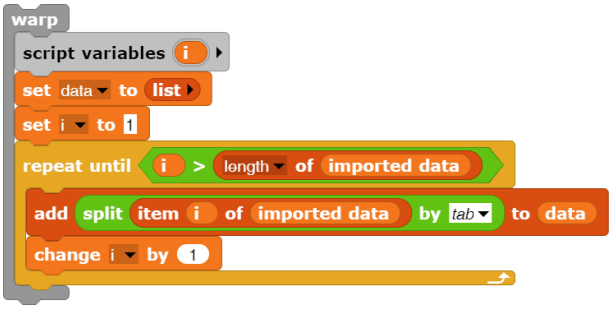


[73] Mit Genehmigung des Autors, verfügbar unter snap.berkeley.edu/run#present:Username=jens&ProjectName=SnapMinder

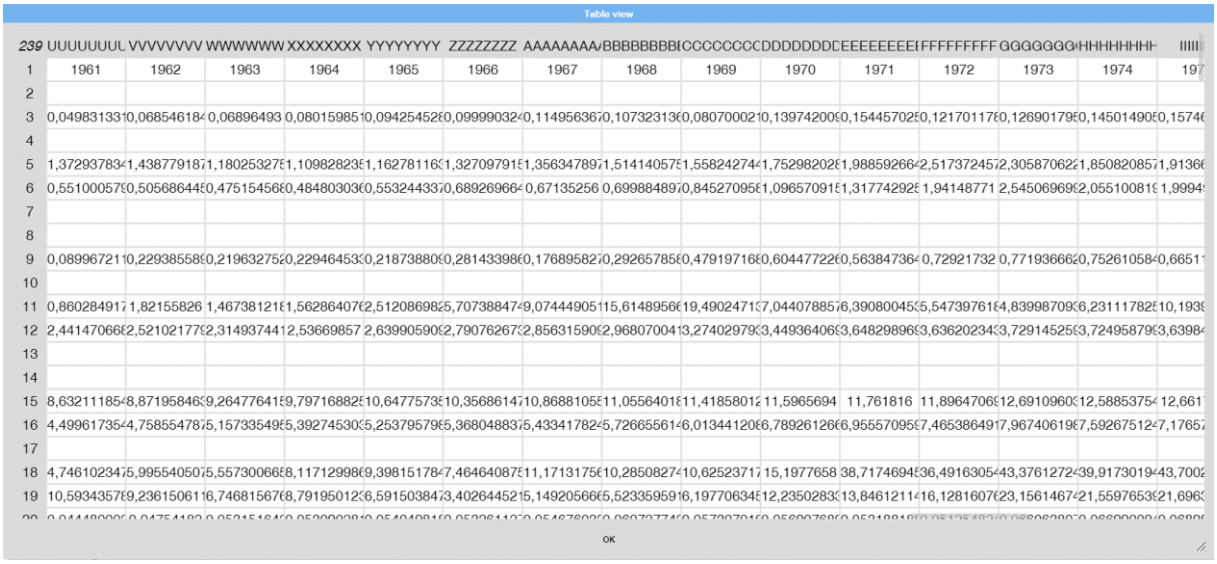
[74] https://www.gapminder.org/

[75] CDIAC: Carbon Dioxide Information Analysis Center

Jede Zeile enthält wieder eine Zeichenkette mit den Daten pro Land, wobei die Daten durch Tabulatoren getrennt sind. Deshalb „zerhacken“ wir die Liste zeilenweise auf die gleiche Art, aber mit einem anderen Trennzeichen, und fügen die Teillisten einer neuen Listenvariablen namens *data* hinzu.



Damit stehen die erforderlichen Daten zur Bearbeitung in *Snap!* bereit.



Die SnapMinder Daten

Das Programm enthält die erforderlichen Daten in der oben beschriebenen Form in den Variablen *income data*, *life data* und *population data*. Diese bereitet es mithilfe von Listenoperation höherer Ordnung für den weiteren Gebrauch auf[76]. Als Beispiel zeigen wir die Berechnung der Bevölkerungsdaten:

Die *population data* werden in eine Liste umgewandelt (hier in „einem Schritt“), wobei die „ohne interessanten Inhalt“ rausgeworfen werden.

Die Operationen sind durch ihre Verschachtelung sehr kompakt. Nimmt man sie auseinander, dann sind sie aber gut verständlich. Als Beispiel nehmen wir den ersten verschachtelten Block. Er kann „von hinten“ gelesen werden als …

Daten der vorhandenen Länder wie oben besprochen in eine Tabelle verwandeln,

unbrauchbare Daten („.. keine Zahlen“) aussortieren und

das Ergebnis der Variablen *population* zuweisen.

population

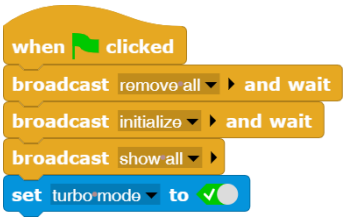
194	A	B	C	D	E	F
1	Total population	1800	1810	1820	1830	1840
2	Afghanistan	3280000	3280000	3323519	3448982	3625022
3	Albania	410445	423591	438671	457234	478227
4	Algeria	2503218	2595056	2713079	2880355	3082721
5	Andorra	2654	2654	2700	2835	3026
6	Angola	1567028	1567028	1597530	1686390	1813100
7	Antigua and Barbuda	37000	37000	37000	37000	37000
8	Argentina	534000	534000	570719	686703	873747
9	Armenia	413326	413326	423527	453507	496835
10	Aruba	19286	19286	19555	20332	21423
11	Australia	351014	342440	334002	348143	434095
12	Austria	3205587	3286650	3391206	3538286	3728381

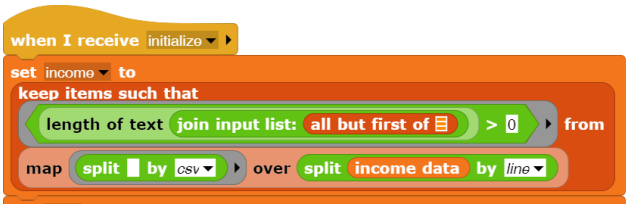
[76] Dabei nutzt Jens Mönig einen kleinen Trick: Schiebt man den Block einer Listenoperation über den *join*-Block aus den Zeichenkettenoperationen, der „leer“, also ohne Eingabeparameter dargestellt wird 

, dann verwandelt sich der in den *join input list*-Block 

, der die Liste in einen einfachen String umwandelt. Die Funktion lässt sich auch leicht selbst schreiben.

Das Programm startet mit drei Nachrichten, die veranlassen, dass alte Länder-Sprites sich selbst löschen und die anderen Objekte, insbesondere die Datenlisten, initialisiert werden. Bei den Daten bewirkt das:





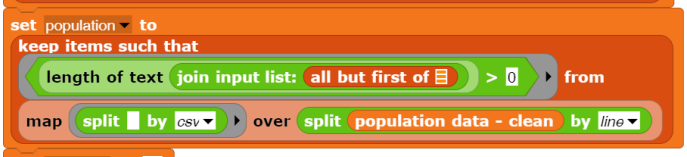
Daten wie eben beschrieben aufbereiten. Zuerst das Einkommen, …

… dann die Lebenserwartung, …

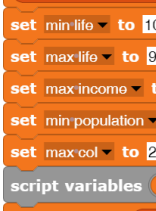
… und daraus die Länder extrahieren.

Das Einkommen den Ländern zuordnen.

Dasselbe für die Lebenserwartung.

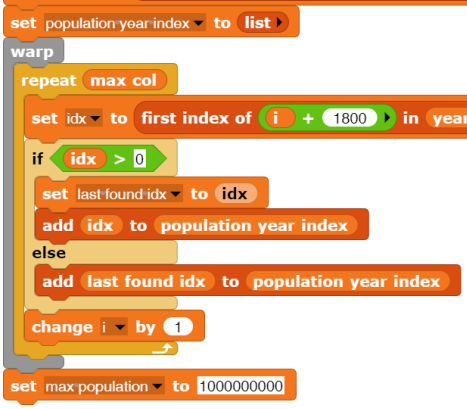


Die Bevölkerungsdaten aus der Hilfsvariablen zurückschreiben.



Einige Variablenwerte setzen.

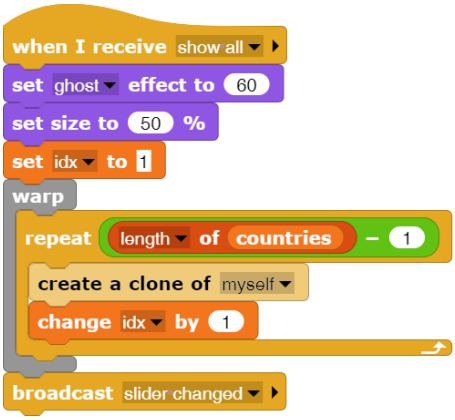
Die Jahreszahlen extrahieren.



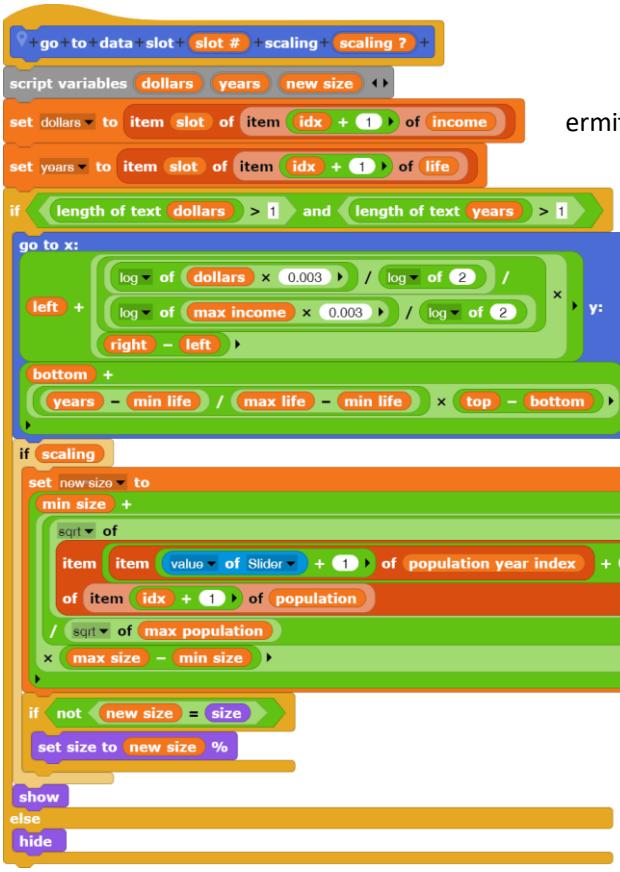
Eine Liste mit Jahreszahlen als Index erzeugen.

Die SnapMinder Länder

Zum Start des Programms werden so viele Klone des Prototyp *Country*, dargestellt durch ein teiltransparentes Rechteck, erzeugt, wie Länder in der Länderliste *countries* enthalten sind. Jeder Klon hat seinen eigenen Index *idx*.



Die wesentliche Funktion der Länder ist es, sich im Koordinatensystem aus mittlerem Einkommen und Lebenserwartung in Abhängigkeit vom betrachteten Jahr zu positionieren. Dazu …

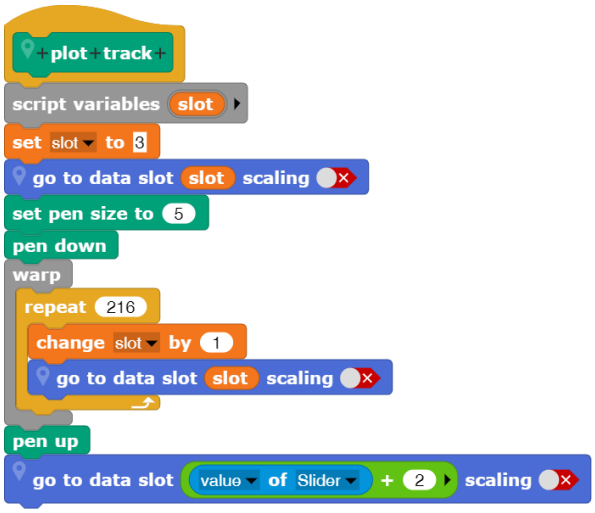


ermitteln sie diese Daten für ihr Land, …

… bestimmen daraus die Position …

… und ihre Größe, die durch die Bevölkerungszahl des Landes im betrachteten Jahr gegeben wird.

Dieser Block wird u. a. aufgerufen, wenn ein *Plot* des Landes, also die Bewegung im Koordinatensystem mit dem Jahr als Bahnparameter, erzeugt wird.



SnapMinder benutzen

Die Darstellung ist beeindruckend, weil sich einerseits die Länder im Laufe der Zeit von links-unten nach rechts-oben bewegen, also eine positive Entwicklung nehmen. Betrachtet man aber einige Länder genauer, dann ist diese Entwicklung durchaus nicht kontinuierlich: es gibt abrupte Ausschläge nach unten, Rückwärtsbewegungen, Kreise, periodische Bewegungen, … Das Programm gibt Anlass, über die Ursachen dieser Entwicklungen zu recherchieren, und da stößt man auf einige Überraschungen! Wir zeigen Plots einiger Länder, recherchieren sollten dann Sie! 😉

USA

Deutschland

China

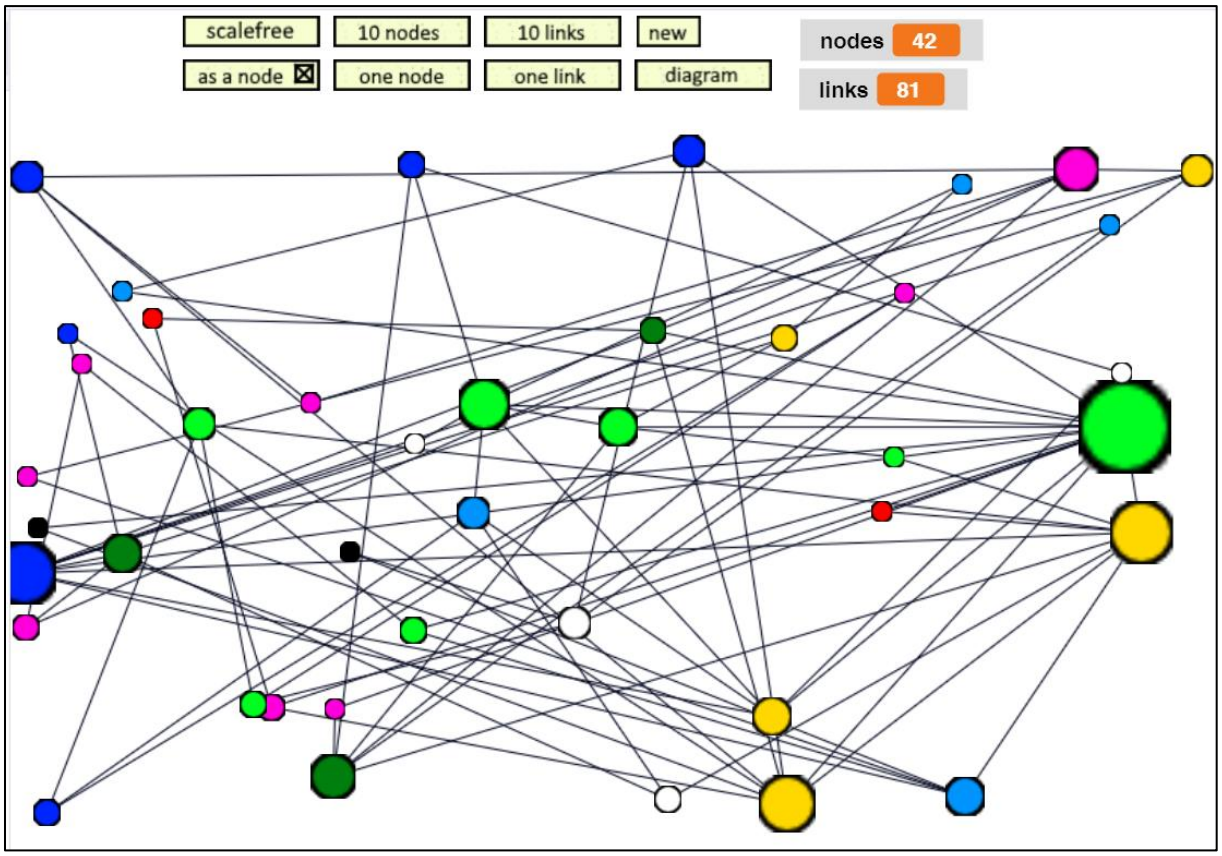
Indien

Norwegen

Somalia

17.3 Konnektiviät: die Welt ist klein[77]

Altersstufe: *Sekundarstufe II* Material: *Connectivity*



Die Behandlung von Netzwerken wird oft auf Protokolle und andere technische Details reduziert. Man kann aber auch andere Fragen stellen, z. B. nach dem Zusammenhang von Netzen.

- Wenn wir *n* Knoten haben, wie viele Verbindungen (Links) benötigen wir, damit das Netz weitgehend zusammenhängend ist?
- Oder umgekehrt: Wie viele und welche Knoten müssen zerstört werden, damit ein Netz in seine Teilnetze zerfällt?
- Oder: Welchen mittleren Abstand, gezählt in Links, haben die Knoten eines Netzes voneinander?

Knoten und Links können von sehr unterschiedlicher Natur sein: Es kann sich z. B. um

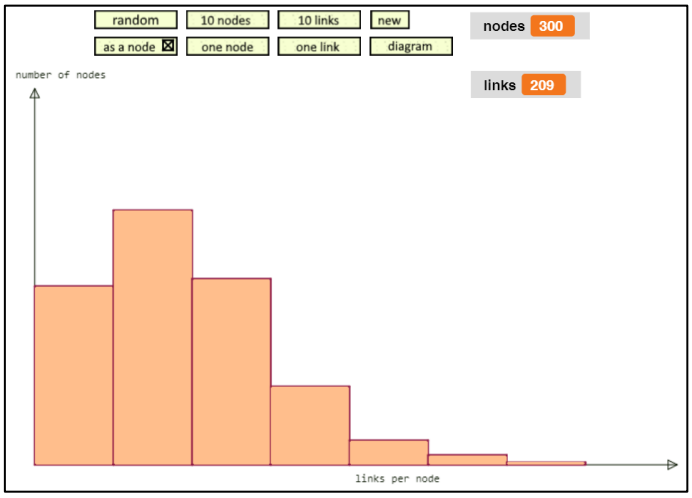
- technische Verbindungen zwischen Informatiksystemen,
- Kunden-/Lieferantenbeziehungen in der Wirtschaft,
- die logischen Verbindungen über verlinkte Webseiten,
- soziale Beziehungen zwischen Personen oder Personengruppen
- Wasserstoffbrücken in organischen Verbindungen,
- neuronale Netze
- oder Infektionsketten handeln.

[77] nach: E. Modrow: Informatik mit Delphi – Band 2, emu-online, 2003

Random Networks

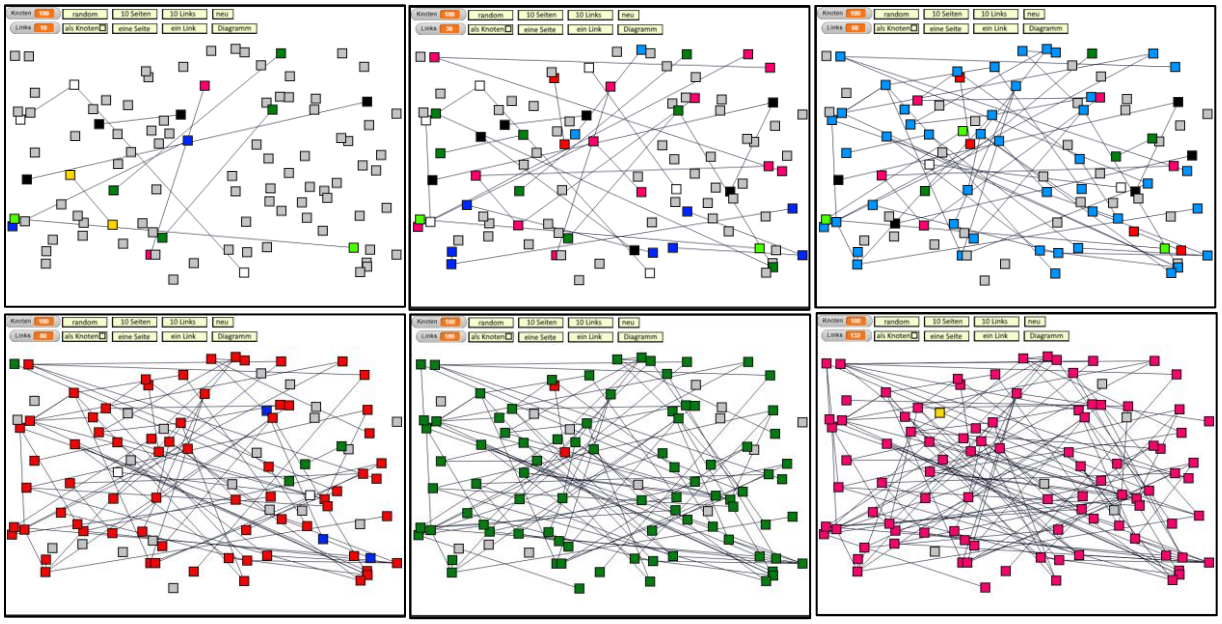
Der Ausgangspunkt für solche Fragestellungen waren *Random Networks*. Sie entstehen, wenn wir *N* Netzknoten (oder Seiten, …) erzeugen, die wir nachträglich untereinander verbinden (verlinken). Betrachten wir als Beispiel einmal das Internet. Wenn sich dort *N* Seiten befinden, die im Mittel über je *k* Links pro Seite verfügen, dann sind mit *n* Mausklicks k^n Seiten erreichbar. Wir können praktisch jede Seite erreichen, wenn gilt: $k^n = N \rightarrow n = \log N / \log k$. Bei *5 Mrd*. Seiten und *k=7* gilt: *n =11,5*. D. h.: mit ca. 12 Mausklicks im Mittel kann man jede Seite dieses Netzes aufsuchen. Ähnliche Überlegungen und praktische Untersuchungen wurden zu sozialen Beziehungen usw. durchgeführt. Sie sind unter dem Namen *Small World Phänomen*[78] zu finden.

Stellt man die Verteilung der Links pro Seite dar, dann erhält man für Random Networks eine Poisson-Verteilung.



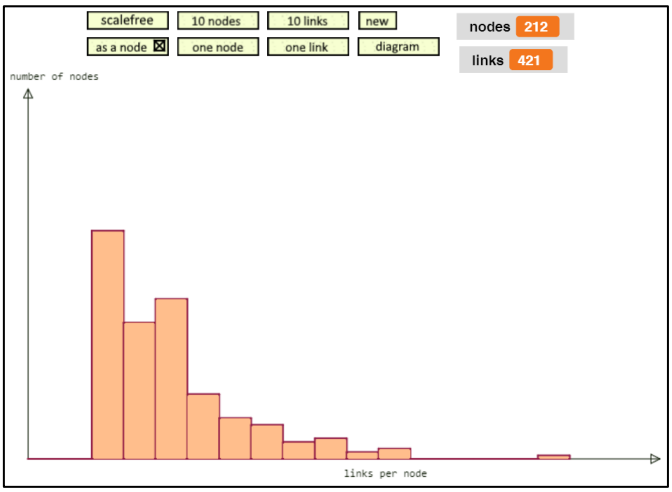
Etwas schwieriger ist es, zu entscheiden, ob ein Netzwerk (weitgehend) zusammenhängend ist, ob also alle Knoten miteinander verbunden sind. Wir können diese Frage durch Färbung beantworten: beginnen wir mit einem Knoten und färben alle von diesem erreichbaren in der gleichen Farbe ein, dann zeigt ein zusammenhängendes Netzwerk eine Art Phasenübergang: fast schlagartig nehmen alle Knoten die gleiche Farbe an.

Man sieht, dass das Netzwerk – bis auf wenige Ausrutscher – zusammenhängend ist, wenn die Zahl der Links in etwa der Zahl der Knoten entspricht. Weitere Links ändern dann wenig daran.



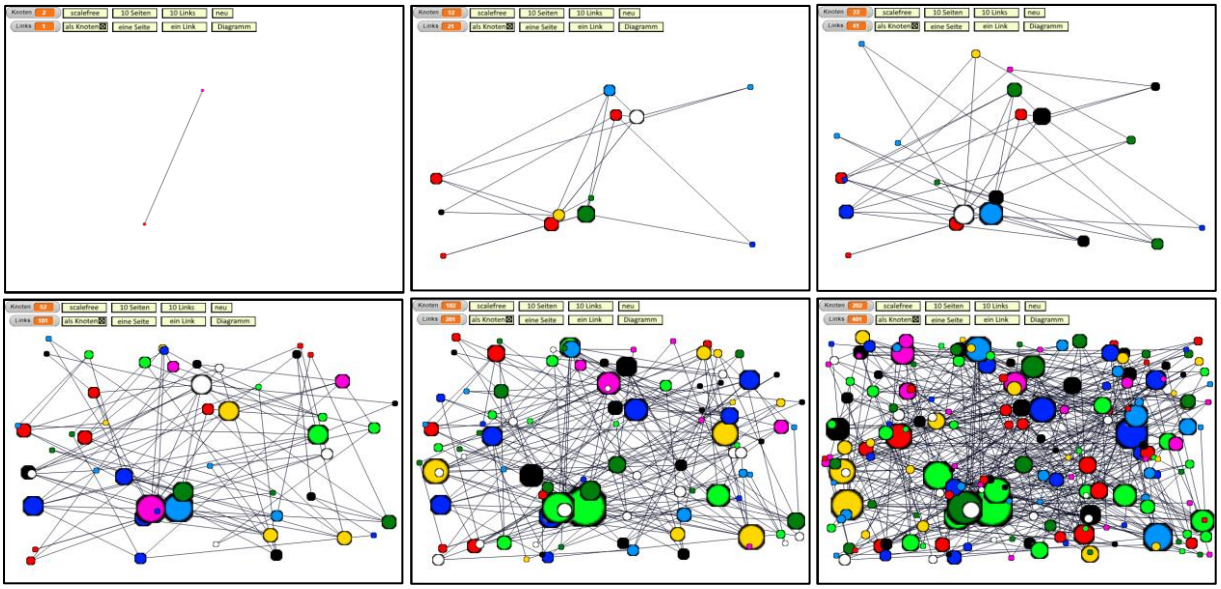
[78] https://de.wikipedia.org/wiki/Kleine-Welt-Ph%C3%A4nomen

Scalefree Networks



Albert-László Barabási zeigte 2002 [79], dass wachsende Netze wie das Internet eine andere Verteilung der Links pro Knoten haben als Random Networks. Sie kann durch eine *Pareto-Verteilung* beschrieben werden. Darstellungen dazu findet man unter *http://barabasi.com/f/623.pdf* bzw. *http://barabasi.com/f/624.pdf.*

Ein durch das abwechselnde Hinzufügen von Knoten und Links entstehende *Scalefree Network* kann man erzeugen, indem man an zwei verlinkte Knoten immer wieder Knoten mit zwei Links zu bestehenden Knoten hinzufügt. Die älteren Knoten haben dann eine höhere Wahrscheinlichkeit, verlinkt zu werden, als die jüngeren. Da das Netzwerk zusammenhängend ist, erübrigt sich eine Färbung zusammenhängender Knoten. Dafür wollen wir die Größe der Knoten von der Zahl ihrer Links abhängig machen.



Scalefree Networks sind auf allen Skalen ähnlich, d. h. zahlreiche Knoten mit wenigen Verbindungen sind mit wenigen Knoten mit vielen Verbindungen, sogenannten *Hubs*, verbunden. Die Verbindungen zwischen Knoten laufen normalerweise vom Startknoten zum nächsten Hub, dann über wenige weitere Hubs zum Zielknoten. Bei Hubs kann es sich z. B. um Personen mit vielen Kontakten (Lehrer/innen, Vertreter, …), zentrale Rechner oder Auslieferungslager in der Warenwirtschaft handeln.

Scalefree Networks sind extrem robust gegenüber technischen Störungen. Fällt z. B. *zufällig* eine Netzwerkverbindung aus, dann betrifft das wahrscheinlich keinen Hub, und wenn doch, gleichen andere Hubs den Ausfall aus. Sie sind allerdings auch extrem anfällig gegen *gezielte* Störungen. Werden nur wenige Hubs in diesem Netzwerktyp zerstört, dann zerfällt das Netz in seine Einzelteile.

Das Thema eignet sich als Einstieg in Diskussionen über Impfschutz, die Verhinderung der Ausbreitung von Krankheiten, der Beeinflussung der politischen Meinungsbildung, der Optimierung von Warenströmen, …

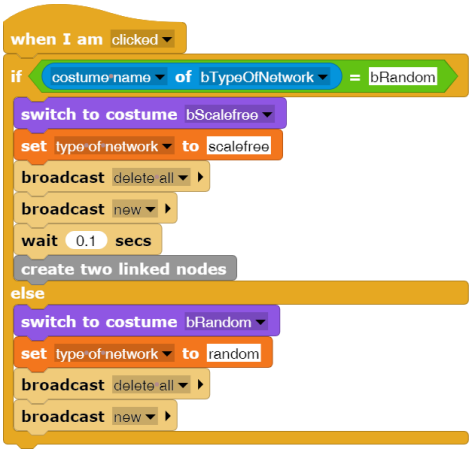
[79] A.Barabási: Linked: the new science of networks, Perseus Publishing 2002

Die Implementierung

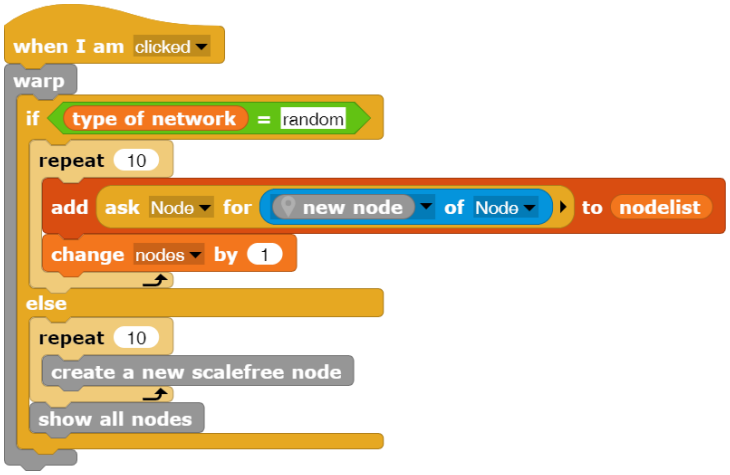
Wir wollen ein ziemlich einfaches Modell als Werkzeug zur Erforschung der Netzwerkeigenschaften erstellen. Es basiert im Wesentlichen auf einem *Knoten*, von dem Klone erzeugt werden, und zwei Listen, von denen die *Knotenliste* die schon erzeugten Knoten enthält und die *Linkliste* aus Teillisten mit den Nummern der beiden Endknoten der Links besteht. Mit deren Hilfe können weitgehend unabhängig voneinander Methoden implementiert werden, die durch die dargestellten Bedienelemente genutzt werden. Die Bedienelemente arbeiten in Abhängigkeit vom gewählten Netztyp (*random/scalefree*) und der Darstellung der Knoten (*rechteckig/rund mit unterschiedlicher Größe*).

12	items
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	

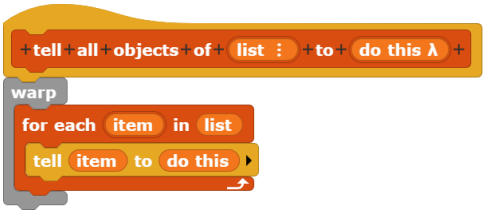
13	A	B
1	9	7
2	4	6
3	8	3
4	9	11
5	1	3
6	3	9
7	6	3
8	10	4
9	6	2
10	8	2
11	10	2
12	1	8
13	5	11



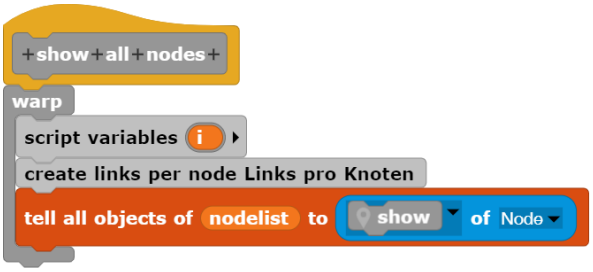
Die Buttons zum Umschalten zwischen den Netzarten bzw. zum Erzeugen von 10 Knoten reagieren auf Mausklicks:



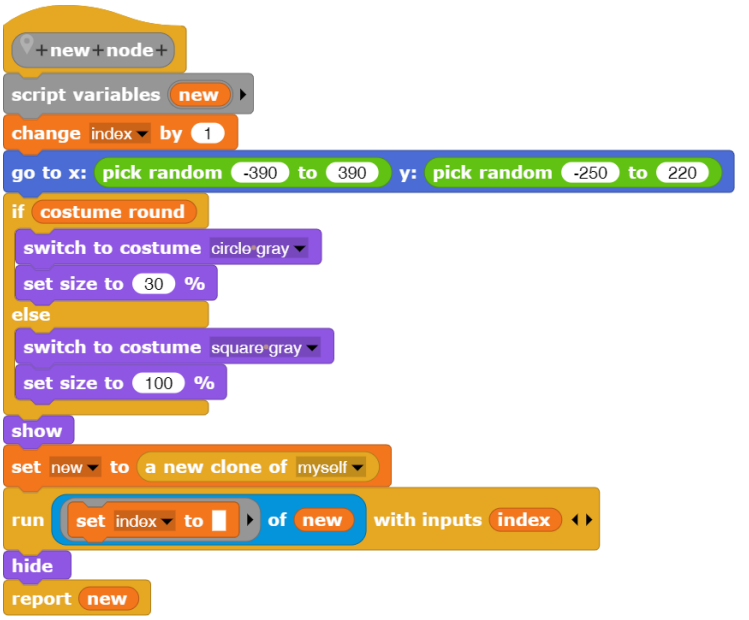
Da wir oft über solche Knotenlisten iterieren müssen, führen wir einen Block ein, der eine Anweisung für alle Objekte einer Liste ausführt:



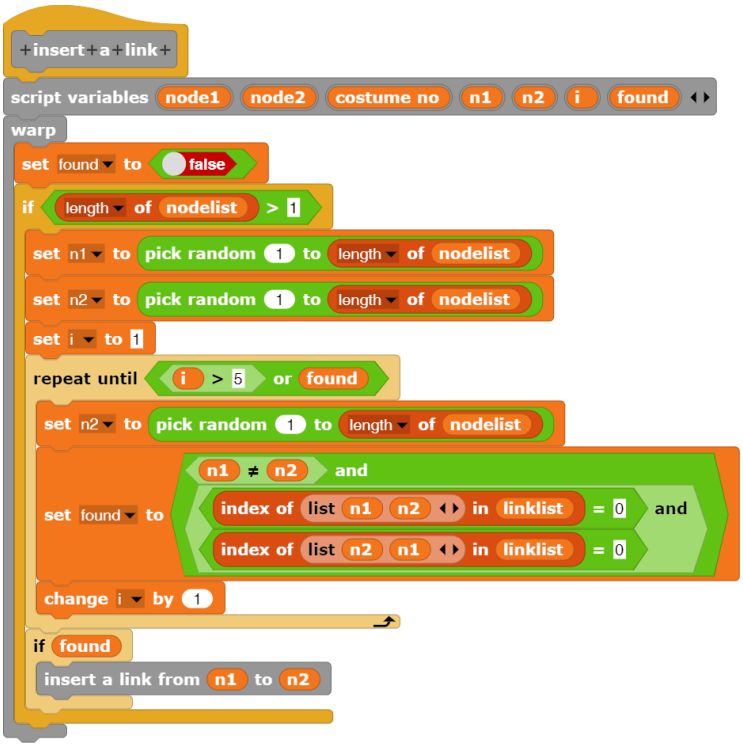
Damit ist dann z. B. die Darstellung aller Knoten sehr einfach:



Neue Knoten werden durch Klonen des Prototyps erzeugt. Der Prototyp kann dann gebeten werden, diese Aktion auszuführen.



Ein neuer Link wird in das Netz eingefügt, indem probiert wird, zwei noch nicht verbundene Knoten zu finden. Dafür muss die Linkliste daraufhin durchsucht werden, ob der Link schon vorhanden ist. Falls nicht, ergibt die Suche *0*. Damit können die Enden des Links bestimmt werden. Da die entstehenden Netze schnell groß werden, wird nicht allzu lange gesucht.



Ist erstmal bekannt, welche Knoten von einem Link verbunden werden sollen, …

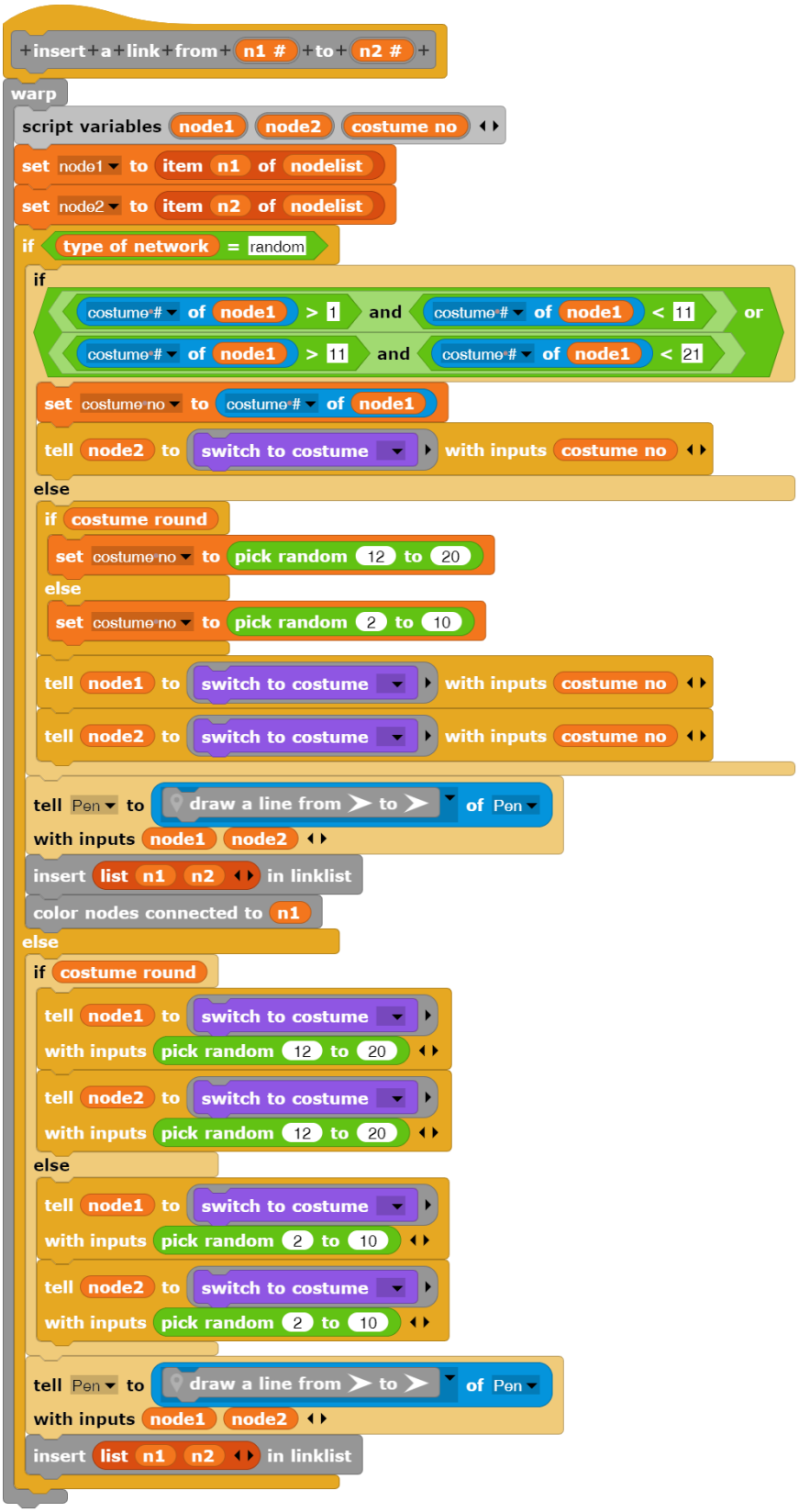
… dann werden die betroffenen Knoten gesucht, …

… je nach Netztyp das Kostüm ausgewählt und die Knoten gebeten, dieses anzunehmen.

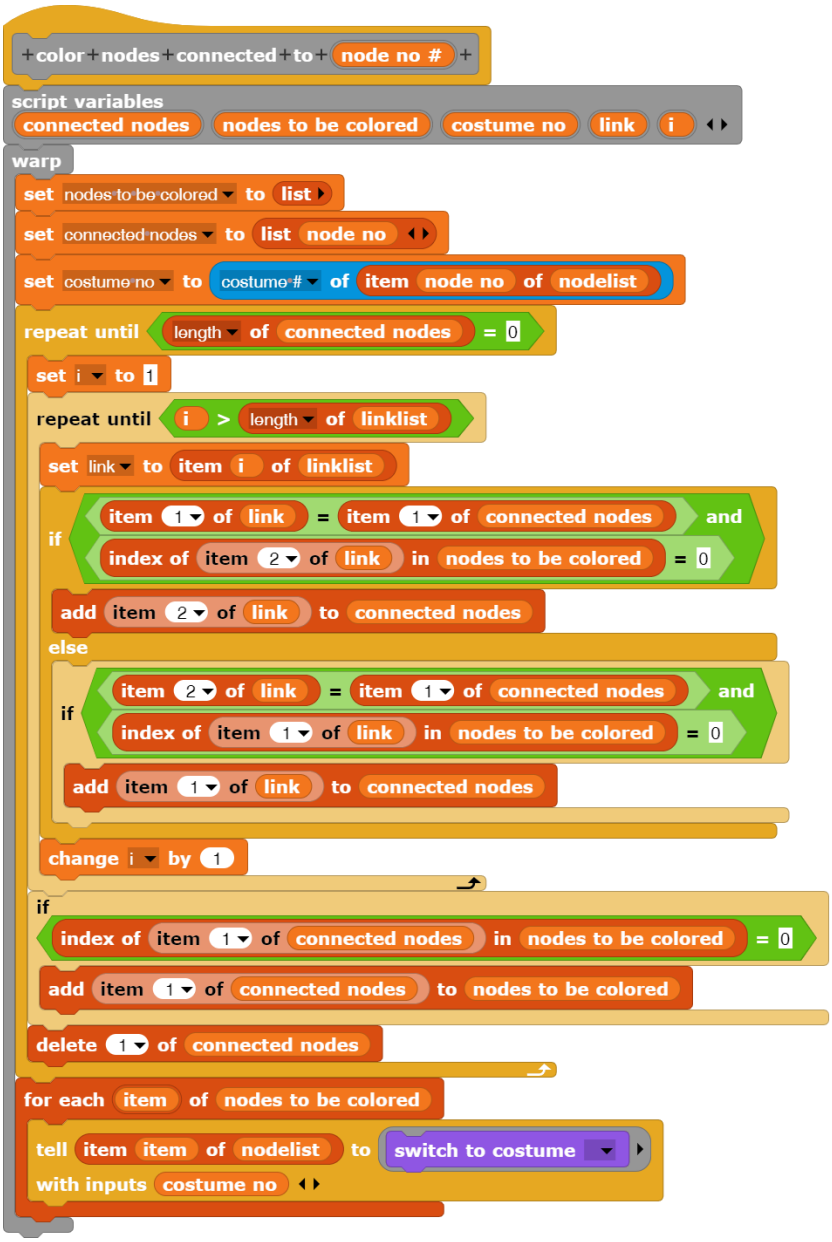
Der Stift wird gebeten, eine Linie zwischen den Knoten zu ziehen.

Zuletzt wird der neue Link in die Linkliste eingetragen und die zusammenhängenden Knoten werden gleich gefärbt.

Bei Scalefree Networks ist es etwas einfacher, weil die Kostüme nur zufällig gewählt werden.



Das Einfärben der zusammenhängenden Teilnetze ist der aufwändigste Teil. Wir arbeiten mit zwei Listen, von denen die *connected nodes* alle Knoten erhält, die man vom Ausgangsknoten aus erreichen kann. Die *nodes tob e colored* nehmen die Knoten auf, die gefärbt werden müssen.



Wir beginnen mit der übergebenen Knotennummer als Anfang und merken uns dessen Kostüm.

Solange noch Knoten in der Liste sind, untersuchen wir die Linkliste, ob die erste Knotennummer der verbundenen Knoten in dem Link entweder links oder rechts auftaucht. Falls ja, ist der andere Knoten auch mit dem Ausgangsknoten verbunden und wird in die Liste aufgenommen, wenn er noch nicht drin ist.

Falls der erste Knoten der Liste noch nicht in der mit den zu färbenden Knoten enthalten ist, wird er dort eingetragen und aus der Liste der verbundenen Knoten entfernt.

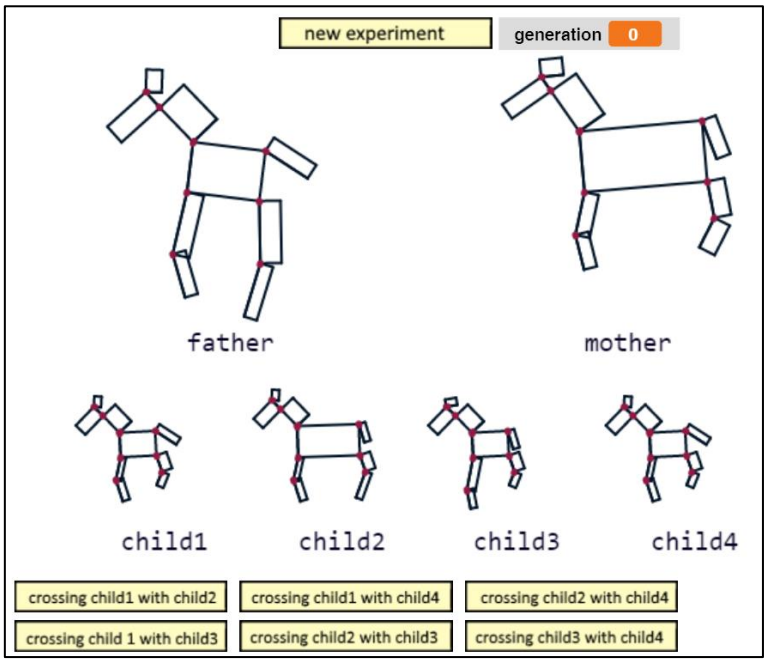
Zuletzt werden die Kostüme aller zu färbenden Knoten auf den gleichen Wert, den der Kostümnummer des Ausgangsknotens, gesetzt.

Die Bedienelemente, die beiden (und weitere) Netztypen, das Erzeugen, Verbinden und Färben der Knoten sowie die Diagrammerstellung basieren auf den Teillisten und sind weitgehend unabhängig voneinander bearbeitbar. Das Thema eignet sich deshalb gut für arbeitsteiliges Vorgehen.

17.4 Evolution

Altersstufe: *Sekundarstufe II*
Material: *Evolution*

Das Ziel dieses kleinen Projekts ist es, mit möglichst einfachen Methoden ein vorzeigbares Ergebnis zu produzieren, das bei Bedarf auch im Unterricht genutzt werden kann. Die Methoden z. B. zur Darstellung der Tiere werden teilweise durch „Rumprobieren“ gefunden, was natürlich Verbesserungen herausfordert. Das soll auch so sein. Die Ansatzpunkte der Teile sind in den Bildern etwas hervorgehoben gezeichnet.



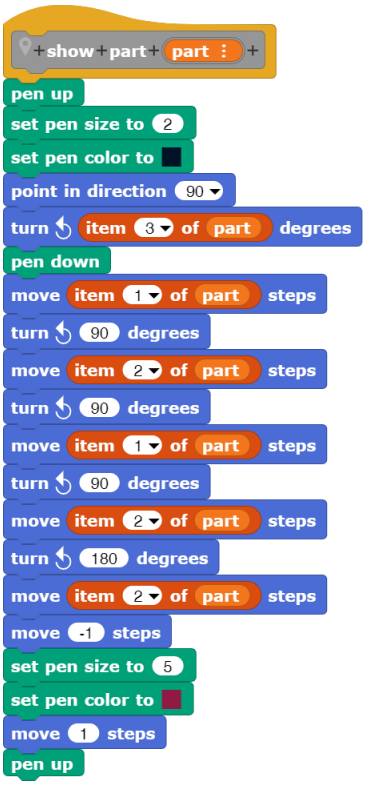
Im Projekt werden „Tiere“ zufällig erzeugt, die aus je 9 Rechtecken zufälliger Größe bestehen, die so gedreht werden, dass eine Art Pferd entsteht. Bei anderer Zusammensetzung lassen sich schnell andere „Tiere“ erzeugen. Die Teilrechtecke werden immer in gleicher Reihenfolge und Orientierung gezeichnet, sodass ausprobiert werden muss, wo das Zeichnen jeweils zu beginnen hat. Dieses Problem lässt sich mit etwas Mathematik natürlich auch eleganter lösen, und wenn man mithilfe von Parametern darauf einwirken kann, wie ein Rechteck zu zeichnen ist, dann geht es – auf andere Art – auch schöner. Aber es geht eben auch ganz einfach.

Nach der Erzeugung zweier Tiere werden vier Nachkommen geschaffen und etwas kleiner darunter dargestellt. Aus diesen kann man zwei auswählen und zu neuen Eltern ernennen. Macht man das wiederholt, dann kann man bestimmte Merkmale „herauszüchten“, z. B. kleine Köpfe oder kurze Beine. Bei jedem Kreuzungsvorgang werden die Merkmale zufällig etwas verändert. Wird ein Teil zu klein, dann fällt es weg. Man kann also aus den anfänglichen Pferden so etwas wie Robben oder Strauße züchten.

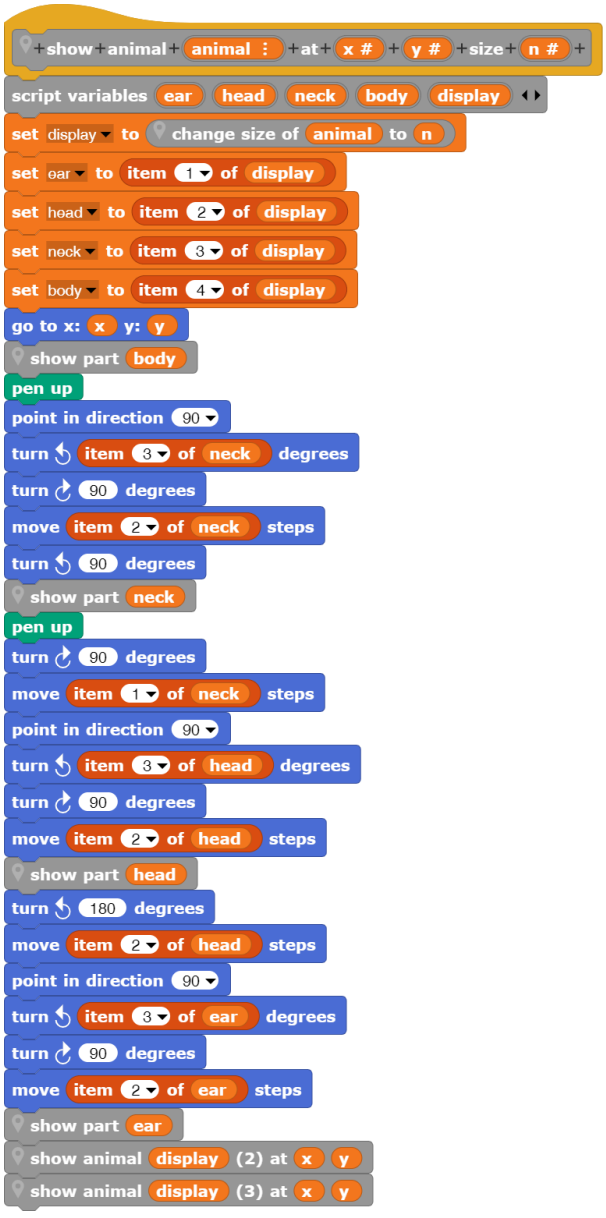
Es bietet sich an, durch Mutationen auch mal neue Teile entstehen zu lassen oder den Ansatzpunkt der Teile zu ändern, sie also „wandern“ zu lassen. Für so etwas müssen die Datenstrukturen geändert werden, etwa durch Aufnahme der Koordinaten der Ansatzpunkte und entsprechende Anpassung der Methoden.

Neue Tiere können von dem Objekt *Animal* erzeugt werde, das hat dafür eine lokale Methode. In dieser werden die Teile des Tiers als Listen von „halbwegs brauchbaren“ Zufallszahlen erzeugt. Danach werden sie zu einer Gesamtliste zusammengesetzt.

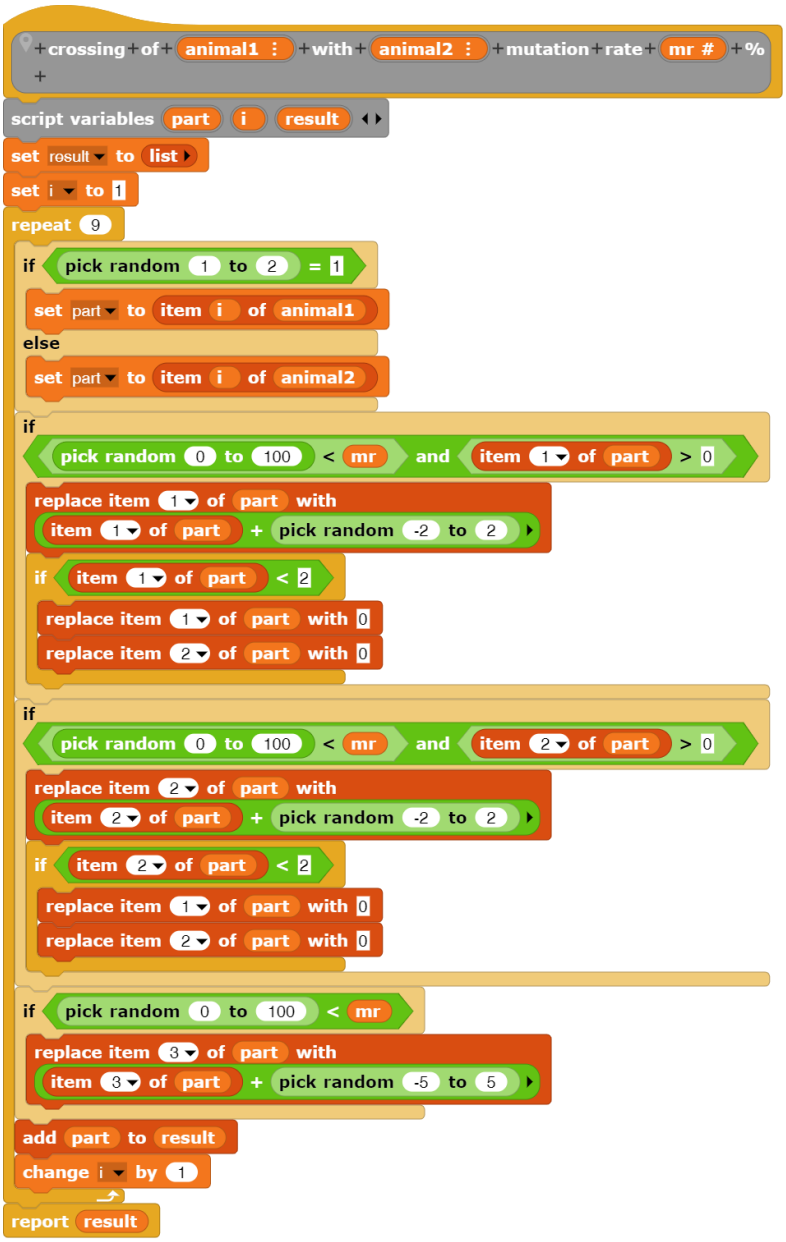
Die Teile der Tiere werden mit immer der gleichen Methode *show part* gezeichnet. Der Stift geht in die horizontale Lage und dreht sich dann auf den in der Liste als drittes Element übergebenen Winkel, danach zeichnet er ein Rechteck mit den als erstem und zweitem Element übergebenen Längen. Zusätzlich wird der Ansatzpunkt etwas hervorgehoben.



Die Methode *show animal* verändert zuerst die Größe des Tiers wie angegeben. Danach werden an den „ausprobierten“ Stellen die Teile gezeichnet. Gezeigt wird nur der erste Teil davon.



Zwei Tiere werden „gekreuzt“, indem zufällig die Teile des einen oder anderen Tiers zu einem neuen zusammengesetzt werden. Bei jedem dieser Vorgänge werden die Maße zufällig verändert – in Abhängigkeit von der benutzten Mutationsrate *mr*.



Auswählen, von welchem Tier ein Teil übernommen wird.

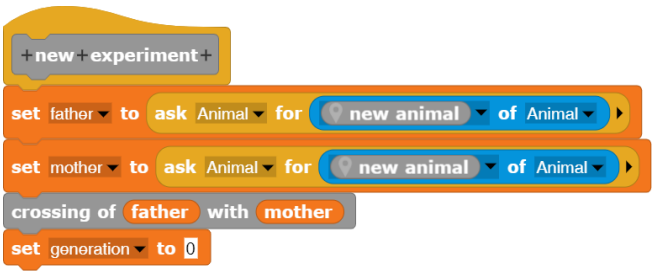
Größe der Breite des Teils zufällig ändern.

Zu kleine Teile fallen weg.

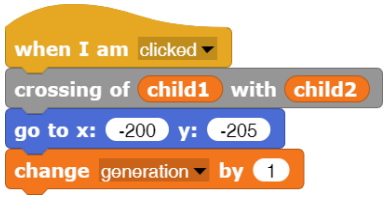
ebenso für die Höhe

Teil bei neuem Tier hinzufügen.

Ergebnis zurückgeben.

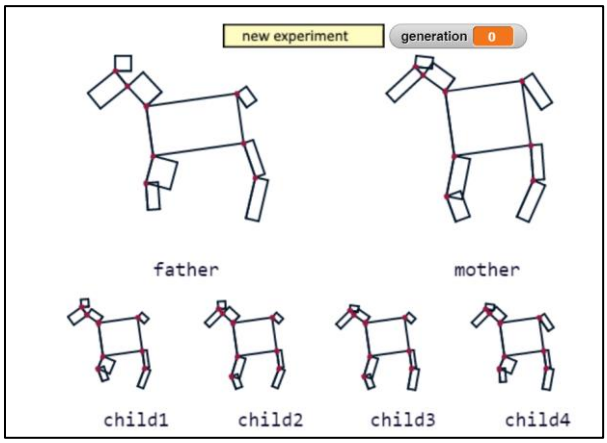


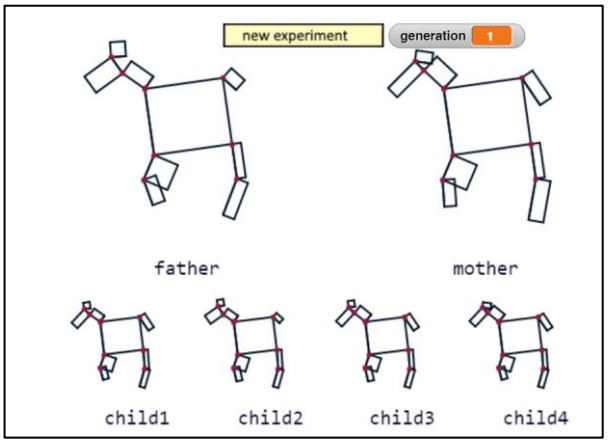
Ein *neues Experiment* wird begonnen, indem das *Animal*-Objekt gebeten wird, zwei neue Tiere als Vater und Mutter zu erzeugen. Die werden dann gekreuzt.

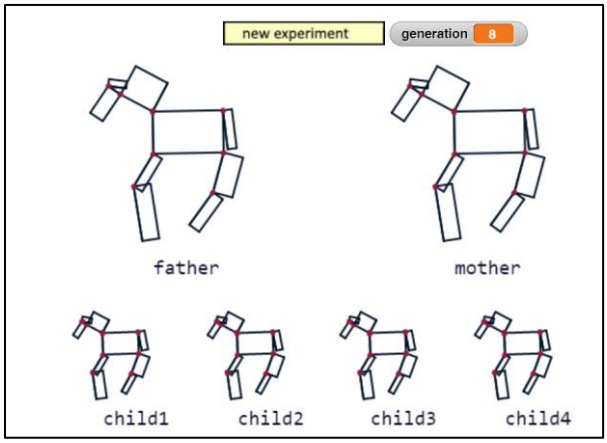


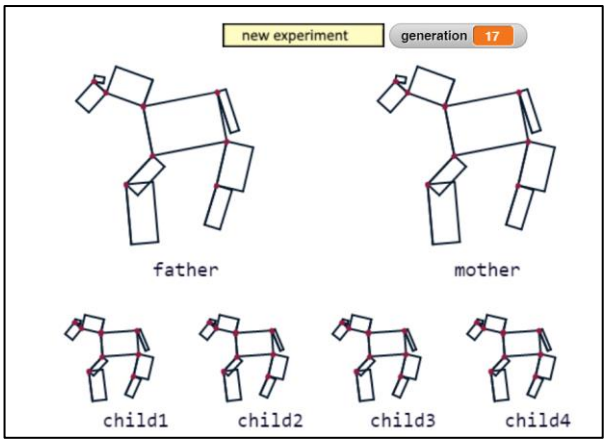
Entsprechend geschieht dieses mit den Kindern.

Wir wollen einmal versuchen, „springende Ponys“ mit kleinen Ohren zu züchten. Dazu erzeugen wir zuerst die Eltern und wählen aus den Nachkommen immer wieder Kandidaten für Ponys aus.





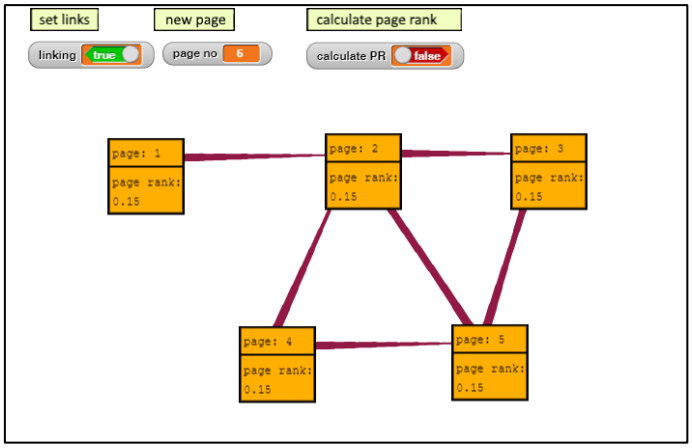




Na ja – die Evolution ist eben unergründlich! 😉

17.5 Webseiten bewerten: PageRank[80]

Altersstufe: *Sekundarstufe II*
Material: *Page rank*



Kennt man die Adressen von Webseiten, dann können wir diese über das Netz direkt erreichen. Was geschieht aber, wenn wir Seiten mit bestimmten Inhalten erst einmal suchen? Für diesen Zweck nutzen wir natürlich die Suchmaschinen, die uns zu bestimmten Stichworten Netzadressen vorschlagen, die sich in ihren Inhaltsverzeichnissen befinden. Diese Verzeichnisse können erstellt werden, indem Webcrawler von Link zu Link automatisch möglichst viele erreichbare Webseiten besuchen und die dort aufgefundenen Stichworte in das Inhaltsverzeichnis der Suchmaschine aufnehmen. Auf diese Weise entstehen meist extrem umfangreiche Adressensammlungen zum gleichen Stichwort.

Da die Benutzer von Suchmaschinen mit großen ungeordneten Adressensammlungen nicht viel anfangen können, müssen die zu einem Stichwort gefundenen Seiten nach ihrer Wichtigkeit geordnet werden. Die Benutzer nutzen dann meist relativ wenige Adressen, die als Erstes erscheinen. Die weit „unten“ stehenden Links werden kaum beachtet. So müssen also zumindest die kommerziell arbeitenden Anbieter im Netz daran interessiert sein, möglichst weit „oben“ in den von Suchmaschinen erstellen Listen aufzutauchen, um von potenziellen Kunden überhaupt gefunden zu werden, und sie nutzen alle Tricks, um das zu erreichen.

Es ist bisher noch nichts über die Bedeutung der Informationen einer Seite für das Stichwort gesagt. Sein Auftauchen alleine bedeutet noch nicht viel. Enthält eine Seite z. B. den Text „Hier steht nichts über Göttingen!“, dann wird sie trotzdem in die Inhaltsverzeichnisse aufgenommen, die das Stichwort „Göttingen“ betreffen. Wir brauchen also andere Bewertungskriterien. Im einfachsten Fall geben die Autoren einer Webseite in den Meta-Tags Stichworte zum Inhalt der Seite an:
<meta name = "keywords" content = "Snap!,Schule,Informatik">

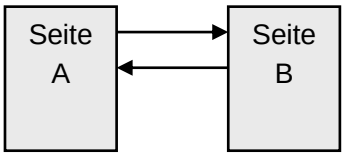
Diese Möglichkeit wird aber oft missbraucht, indem oft verwendete Stichworte – die den Seiteninhalt gar nicht betreffen – angegeben werden, um potentielle „Opfer“ auf die Seite zu lenken. Nicht sehr hilfreich ist auch die Idee, zu zählen, wie oft das Stichwort auf der Seite vorkommt. Für diesen Fall enthalten Webseiten manchmal bestimmte Stichworte „unsichtbar“, z. B. indem das Stichwort sehr oft in weißer Schrift auf weißem Hintergrund geschrieben wird. Man kann Webseiten natürlich auch durch Menschen bewerten und in die Suchverzeichnisse eintragen lassen. Das ist aber eine sehr teure und relativ langsame Art, Verzeichnisse zu erstellen, und außerdem ist so eine Bewertung natürlich subjektiv. Auch ist es oft schwierig, Seiten mit speziellen Inhalten – z. B. aus der Archäologie – einzuschätzen. Im schlimmsten Fall ergibt sich der „Wert“ einer Seite nicht aus ihrem Inhalt, sondern aus dem Betrag, der für die Bewertung bezahlt wurde.

Eine andere Art, einerseits das Fachwissen von Webautoren für die Bewertung von Webseiten zu nutzen und andererseits den Bewertungsvorgang zu automatisieren, wird im so genannten *PageRank*-Verfahren verwirklicht. Im Unterschied zu den Meta-Tags, die die eigene Webseite bewerten, werden die Links einer Webseite auf andere Webseiten als eine auf Fachwissen basierende Abstimmung aufgefasst, mit der Autoren kundtun, dass andere Webseiten interessante Inhalte enthalten. Verweist also jemand auf eine Seite mit physikalischen Inhalten, dann wird der Autor mit hoher Wahrscheinlichkeit etwas von deren Inhalten

[80] nach: E. Modrow: Technische Informatik mit Delphi, emu-online, 2004

verstehen. Da außerdem meist nicht bekannt sein kann, welche anderen Webseiten auf die eigene verweisen, können Web-Autoren dieses Verfahren nur schwer manipulieren.

Das PageRank-Verfahren bewertet nicht alle Links gleich. Es ermittelt für jede ihm bekannte Webseite einen Rang (eben den PageRank), der das „Gewicht“ dieser Seite beschreibt. Dieser Rang wird bei der „Abstimmung“ durch Links auf alle Verweise aufgeteilt, die von der Seite wegführen. Enthält eine Webseite also nur einen ausgehenden Link, dann erhält dieser das ganze Gewicht der Seite, enthält sie zwei, dann wird das Gewicht halbiert usw. (Enthält die Seite gar keinen ausgehenden Link, dann nimmt sie nicht an der Abstimmung teil.) Der Rang einer Webseite steigt also, wenn möglichst viele Seiten mit hohem Rang auf sie verweisen und wenn diese Seiten jeweils möglichst wenige Links enthalten.



Wählen wir als erstes Beispiel zwei Seiten, die wechselseitig auf einander verweisen. Zur Berechnung des PageRanks von Seite *A* – *PR(A)* – benötigen wir den PageRank *PR(B)* von Seite *B*, weil von *B* ein Link zur Seite *A* führt. In die Berechnung von *PR(B)* geht aber wiederum *PR(A)* ein. Wir benötigen also einen alten Wert von *PR(A)*, um den neuen zu bestimmen. Da sich diese Argumentation fortsetzen lässt, muss eine Methode entwickelt werden, um den Einfluss der alten Werte auf die Berechnung des neuen Rangs abklingen zu lassen, damit sich im Laufe der Berechnungen ein stabiles Ergebnis ergibt. Man erreicht dieses, indem man den Beitrag der eingehenden Links mit einem Faktor *d* multipliziert, der kleiner als *1* ist. Da dieser in jede Berechnung eingeht, werden die „sehr alten“ PageRanks mit d^n multipliziert, also einer Zahl, die sich immer mehr der Null nähert. Man wählt z. B. den Wert *0,85* für *d*. Wenn wir die Zeitpunkte, zu denen der PageRank in der Vergangenheit berechnet wurde, mit t_1, t_2, t_3, … bezeichnen, wobei ein größerer Index einen früheren Zeitpunkt bedeuten soll, dann erhalten wir für unsere beiden Webseiten:

$$PR_{t_1}(A) = \ldots + 0{,}85 \cdot PR_{t_2}(B) = \ldots + 0{,}85 \cdot (\ldots + 0{,}85 \cdot PR_{t_3}(A)) = \ldots + 0{,}85 \cdot \ldots + 0{,}85^2 \cdot PR_{t_3}(A) = \ldots$$

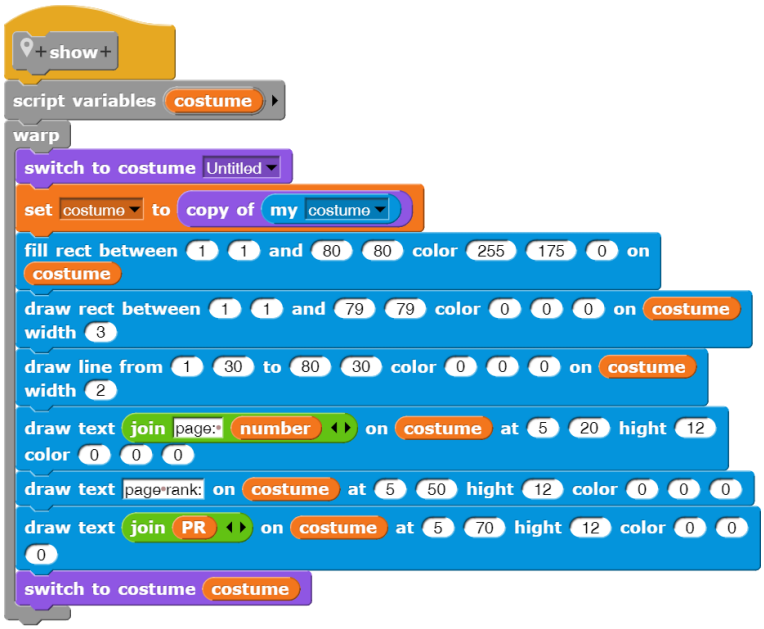
Hätte die Seite *B* mehr als einen ausgehenden Link, dann müssten wir in der Rechnung ihren Rang noch durch die Anzahl der Links – *C(B)* – teilen. Entsprechend müssen wir bei den anderen Seiten vorgehen, die Links auf die Seite *A* besitzen. Bezeichnen wir diese *n* Webseiten mit T_1, T_2, …, T_n und ersetzen die drei Pünktchen in der oben angegebenen Beziehung durch *(1-d)*, dann erhalten wir die Originalformel, die anfangs für die PageRank-Berechnung von Google angegeben wurde:

$$PR(A) = (1-d) + d \cdot \left(\frac{PR(T_1)}{C(T_1)} + \frac{PR(T_2)}{C(T_2)} + \ldots + \frac{PR(T_n)}{C(T_n)}\right), \quad d = 0{,}85$$

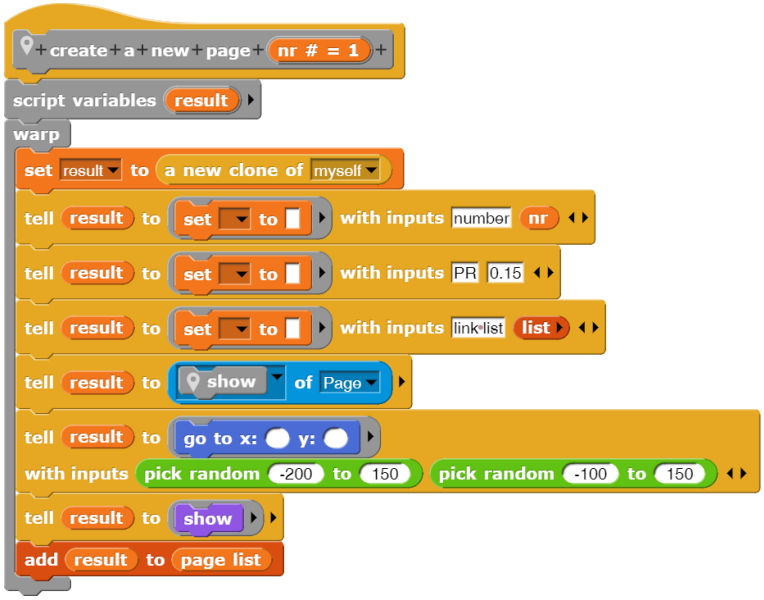
Der Rang einer Webseite beträgt danach mindestens *0,15*. Aber welchen Einfluss haben die anderen Terme? Wir wollen die Frage durch ein Simulationsprogramm klären, in dem symbolische Webseiten erzeugt und verlinkt werden können. Darin können dann die PageRanks berechnet werden.

In unserem Programm benötigen wir neben den abgebildeten Buttons, die zur Steuerung der Funktionalität dienen, den Prototyp einer *„Page“*, bei der es sich (hier) um eine Webseite handeln soll, sowie eine globale Liste aller erzeugten Pages. Jede Seite enthält eine *link list* mit den Nummern der verbundenen Seiten, eine *number*, einen PageRank *PR* und eine Hilfsgröße *PRnew*, in der der jeweils neu errechneten PageRank summiert wird. Pages sollen sich am Bildschirm darstellen können. Da dabei das Kostüm verändert wird, operieren wir besser auf einer Kopie der aktuellen Version. Ein entsprechender Block lässt sich schnell schreiben.

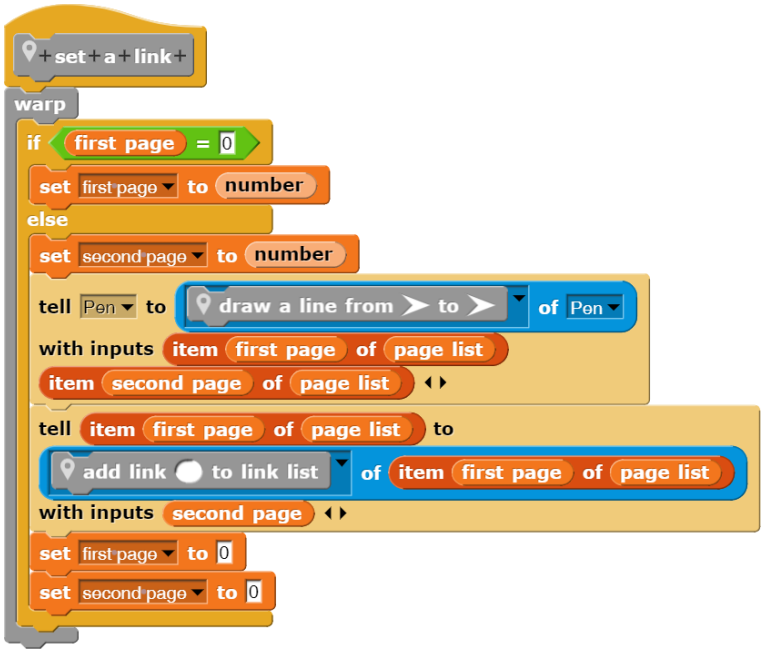
Für die Darstellung von Text und Linien auf dem Sprite benutzen wir wieder die entsprechende Grafikbibliothek.



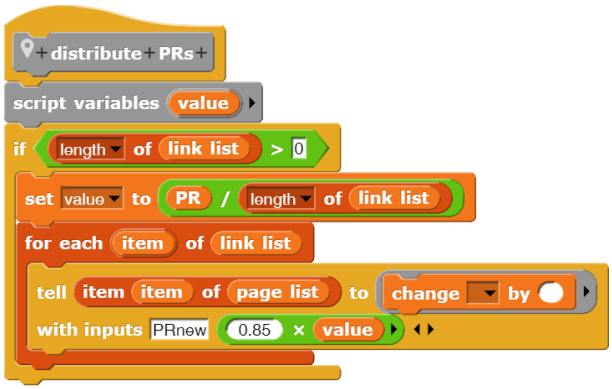
Die wichtigste Aufgabe des Prototyps ist es, Klone seiner selbst zu erzeugen. Wir speichern so einen Klon in einer Skriptvariablen *result* und bitten diese dann durch eine Befehlsfolge, die Operationen auszuführen, die das gewünschte Ergebnis erzeugen. Die erzeugte Seite wird der *Pageliste* hinzugefügt.

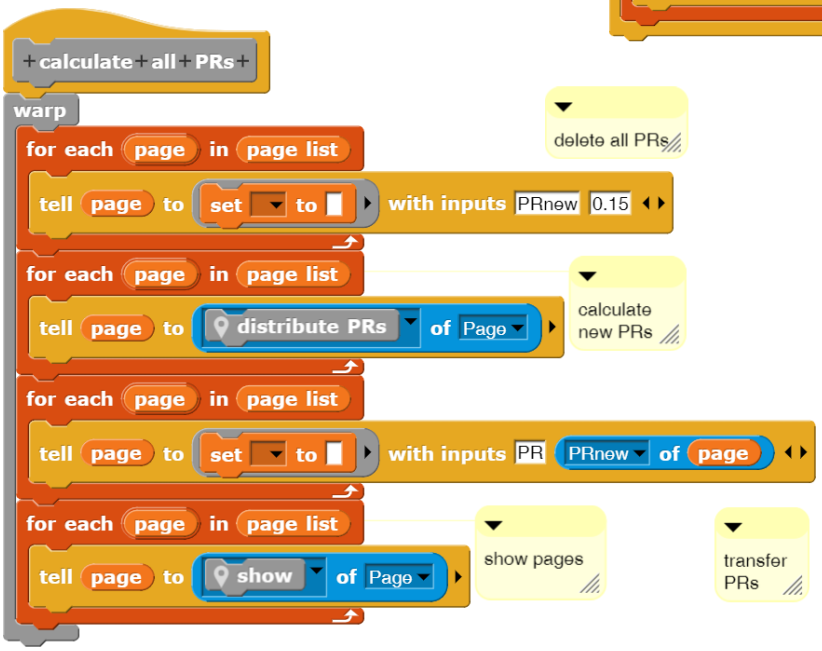


Im entsprechenden Modus werden Pages verbunden, indem nacheinander zwei Seiten angeklickt werden. Die Nummern der betroffenen Seiten werden in zwei globalen Variablen gespeichert. Danach kann die erste gebeten werden, sich mit der zweiten zu „verlinken“. Der *Pen* zeichnet eine in der Dicke abnehmende Linie zwischen den Seiten, eine Art Pfeil. (Gegenseitig verbundene Seiten erhalten damit eine Verbindung fast gleichbleibender Dicke.) Danach wird die zweite Seite in die Linkliste der ersten eingefügt.



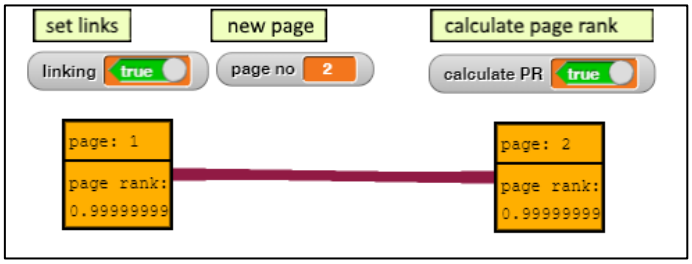
Bei der Neuberechnung der PageRanks muss jede Seite ihren aktuellen Wert *value* an alle verbundenen Seiten verteilen. Die Seite berechnet diesen Wert und bittet alle Seiten der Linkliste, deren Hilfswert *PRneu* entsprechend zu erhöhen.



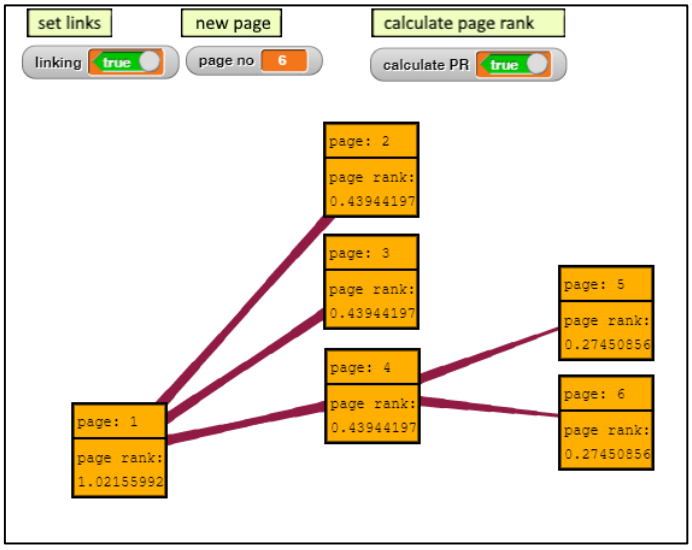


Mit diesen Hilfsmethoden können die Pageranks berechnet werden. Zuerst einmal werden alle Hilfsgrößen der beteiligten Seiten auf Null gesetzt. Danach verteilen alle Seiten ihre Werte an die verbundenen anderen Seiten. Ist das erledigt, dann werden die Hilfsgrößen in die PR-Variablen kopiert und die Seiten mit den neuen Werten neu gezeichnet.

Wir wollen jetzt unser Simulationsprogramm benutzen. Wir erzeugen zwei Webseiten, verlinken sie und berechnen die PageRanks. Man sieht, dass die Werte (übrigens unabhängig vom anfänglichen PageRank) gegen 1 konvergieren. Das ist natürlich keine Überraschung, denn genau dieses haben wir mit der Einführung des „Dämpfungsfaktors“ 0,85 ja beabsichtigt.

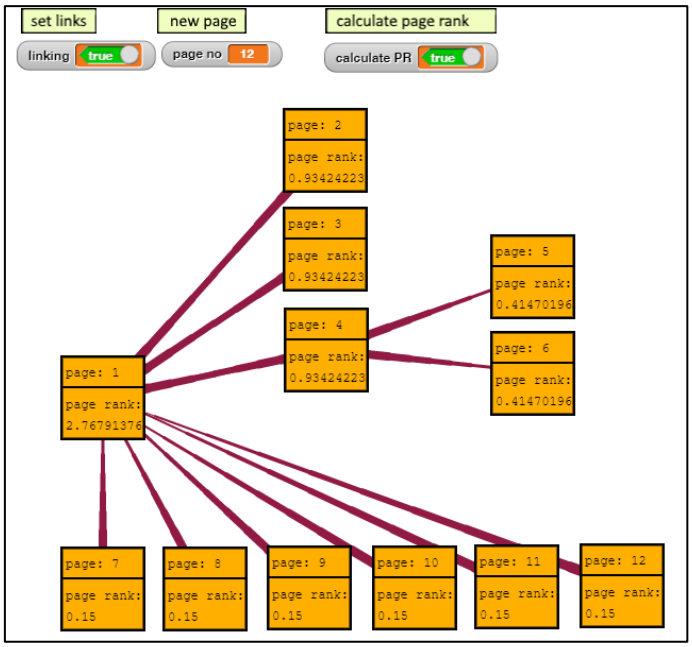


Als nächstes Beispiel wählen wir den Aufbau einer typischen Homepage mit einer Baumstruktur, die von einer Indexseite ausgeht und in Unterverzeichnisse verzweigt.



Wir nehmen jetzt an, dass es zusätzliche externe Seiten gibt, die auf unsere Homepage verweisen.

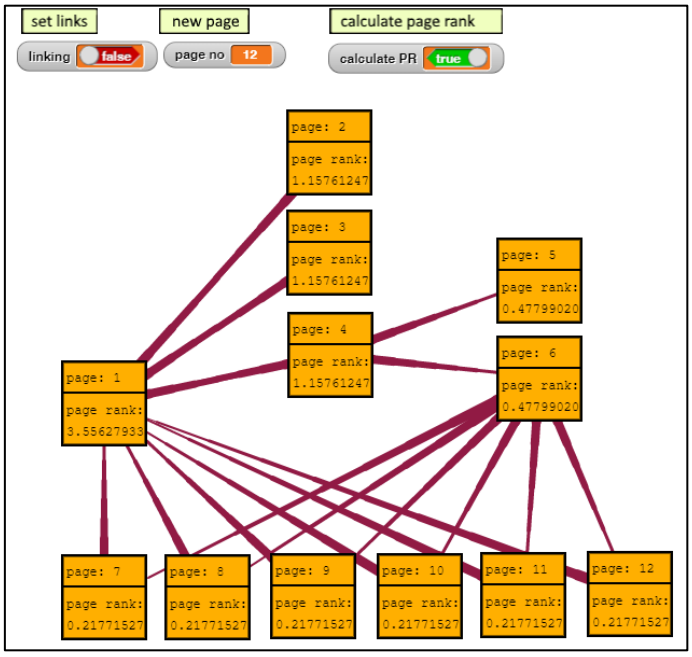
Der PageRank der Homepage steigt erheblich, auch das Gewicht der internen Seiten nimmt zu.



Zuletzt wollen wir annehmen, dass die externen Seiten wiederum in einer Linkliste der Homepage referenziert werden.

Der Rang der Homepage steigt weiter. Man sieht, wie in einem Netz von Seiten, die wechselseitig aufeinander verweisen, um ihre „Hochachtung“ vor einander auszudrücken, die Bedeutung der Seiten wächst.

Beim PageRank-Verfahren handelt es sich erstmal um einen technischen Vorgang, der auch auf andere, z. B. soziale Systeme übertragbar ist[81]. Es führt aber schnell auf gesellschaftspolitische Fragestellungen, weil nicht der Inhalt der Seiten, sondern deren Struktur und Funktionalität im Fokus sind.



1. Wenn das Ergebnis der PageRank-Berechnung entscheidend für die „Sichtbarkeit“ der Seiten ist[82], weshalb dürfen dann kommerziell orientierte Privatunternehmen über diese Sichtbarkeit entscheiden?
2. Die Intelligenz des Systems resultiert aus der Expertise derjenigen, die auf sehr unterschiedlichen Gebieten bewusst Links gesetzt haben. Ist das Ergebnis dann nicht eigentlich ein öffentliches Gut, das allen zur Verfügung stehen sollte, ohne dass einige daraus Gewinn (und Macht) ziehen?
3. Die Suchergebnisse müssten ja eigentlich immer gleich geordnet werden, wenn nur der PageRank entscheidend wäre. Offensichtlich ist das nicht der Fall: die Ergebnisse unterscheiden sich je nach Person, die sucht. Sie werden nach deren von der Suchmaschine angenommenen Interessen gefiltert. Im Extremfall erhält man nur noch die Ergebnisse, die man auch sehen will – oder von denen jemand meint, dass man es will. Die politischen Folgen (Stichwort: „Echokammern“) werden aktuell diskutiert.

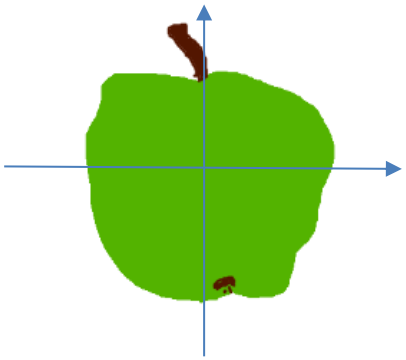
[81] Da kommt es sogar her: https://de.wikipedia.org/wiki/PageRank
[82] Was erst auf den hinteren Plätzen erscheint, ist im Netz praktisch nicht existent.

17.6 Die „intelligente“ Früchtewaage

Altersstufe: *Sekundarstufe II* Material: *Smart scale*

Im nahegelegenen Supermarkt kündigt sich eine Sensation an: die Früchteabteilung hat eine „intelligente“ Waage mit einer Kamera bestellt, die Früchte erkennen und gleichzeitig wiegen soll. Leider wurde nur die Kamera mitgeliefert, die Früchteerkennung muss selbst implementiert werden. Die Früchteabteilung holt sich Hilfe von den Betreuern der Scannerkasse, denn die haben weiter vorne in diesem Buch schon Ähnliches geleistet.

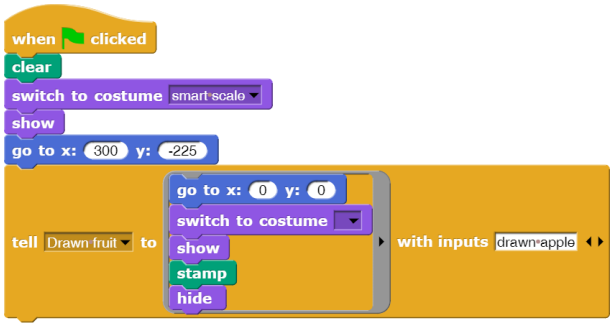
Zuerst versuchen wir einige Kriterien zu finden, mit denen man Früchte unterscheiden kann. Wir zeichnen einen Apfel, eine Orange, eine Aprikose und eine Banane. Die Unterschiede sind offensichtlich:



- Apfel und Orange sind rund, die Banane ist lang
- Orange, Aprikose und Banane sind orange-gelb, der Apfel ist (in diesem Fall) grün
- Die Aprikose ist klein, die anderen sind größer

Aber was bedeuten „rund“, „lang“, „gelb“ und „grün“, „groß“???

Wir wissen es, aber der Computer weiß es nicht. Wir müssen es ihm beibringen.



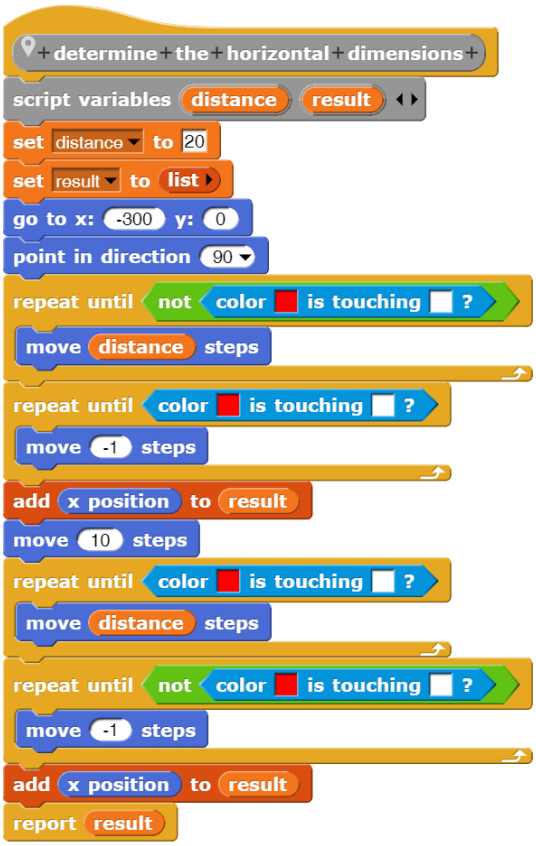
Wir ändern die Bühnengröße auf 800 x 600 Pixel und bringen das Objekt mit den Kostümen der selbst gezeichneten Früchte (*Drawn fruit*) in die Bühnenmitte. Dort bitten wir es, das Kostüm „drawn apple“ anzunehmen und auf die Bühne zu stempeln. Danach soll es sich verstecken.

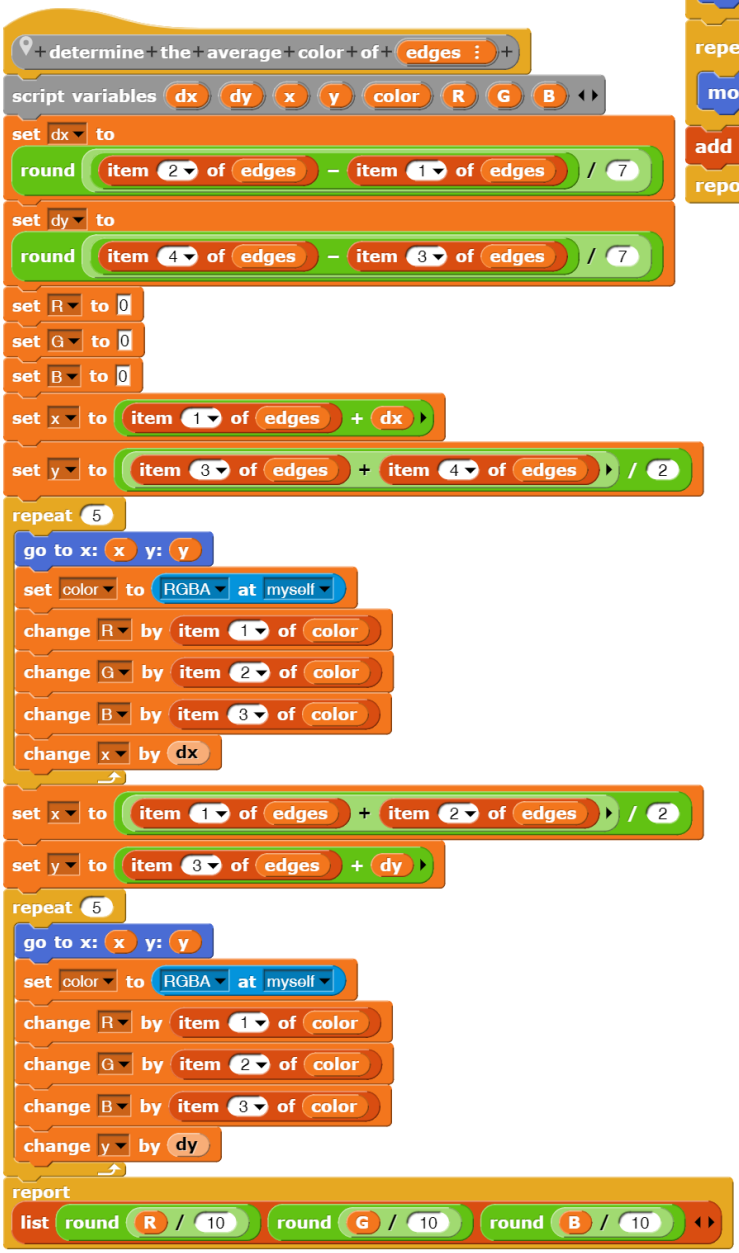
Danach beauftragen wir einen *Laser*, die Eigenschaften der aktuell sichtbaren Frucht zu bestimmen. Dafür soll er, ähnlich wie der Barcodescanner, von links nach rechts und von unten nach oben übers Bild laufen. Dabei misst er die Größe des Objekts auf diesen Strecken und berechnen das Verhältnis der Ergebnisse. „Runde” Objekte sollten ein Verhältnis nahe bei „1“ haben, „lange“ Objekte einen kleinen Wert. Für „ovale“ Objekte sollten wir eigentlich mehrere Messrichtungen benutzen. Aber fürs Erste bedeutet „oval“ für uns „nicht rund und nicht lang“.

Damit das Messen nicht zu lange dauert, macht der Laser so lange größere Schritte, bis er auf die Frucht trifft. Danach geht er in kleinen Schritten zurück zum Rand der Frucht und merkt sich seine x-Position. Das Gleiche macht er am gegenüberliegenden Rand.

Der *determine the horizontal dimensions* - Block des Lasers liefert also eine Liste mit zwei Werten: linke und rechte Grenze. Entsprechend liefert der *determine the vertical dimensions* - Block Unter- und Obergrenze des Objekts. Mit diesen Ergebnissen können wir entscheiden, ob ein Objekt rund, lang oder oval ist. Und wir kennen seine Größe.

Es fehlt noch die Farbe des Objekts. Wir kennen ja schon die Grenzen, innerhalb derer sich die Frucht auf der Bühne befindet. Diese übergeben wir einem Block *determine the average color of* …. In diesem wird der Laser zu 5 Punkten auf der mittleren Horizontalen geschickt, wobei er jedesmal die RGB-Werte bestimmt. Dasselbe geschieht auf der mittleren Vertikalen. Danach bestimmen wir die Mittelwerte der Farbkanäle.





Mit diesen Methoden des Lasers kann dieser die charakteristischen Eigenschaften einer Frucht bestimmen.

3	A	B	C
1	long		
2	big		
3	200	186	1

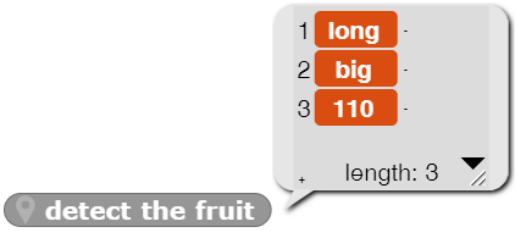
measure the fruit

Normale Früchte haben unterschiedliche Farben. Aber unsere RGB-Werte können 256 * 256 * 256 Farben darstellen, also 16.777.218. Das sind etwas zu viele. Wir benötigen ein Verfahren, um die Farbenzahl zu verringern.

Wir versuchen es so: Für jeden RGB-Kanal entscheiden wir, ob der Farbwert „hoch oder „niedrig“ ist. Falls er hoch ist, setzen wir ihn auf 1, sonst auf 0. Wir erhalten damit nur zwei mögliche Werte für jeden Kanal, also 2 * 2 * 2 = 8 mögliche Farben. Mit diesem Verfahren probieren wir mal aus, ob wir überhaupt noch etwas Sinnvolles sehen können – oder nicht.

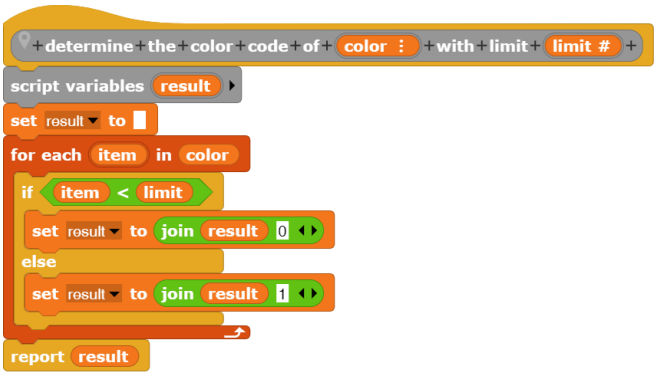
Das sieht doch ganz gut aus!

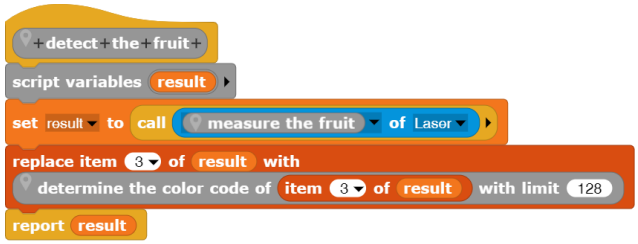
Wir können also die Früchtewaage mit einer Methode ausstatten, die den Laser bittet, die Frucht-Daten zu ermitteln.



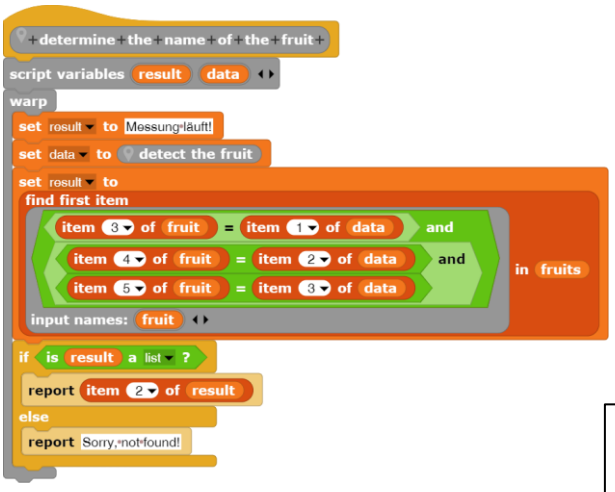
fruits

16	A	B	C	D	E
1	100	apple red	round	big	100
2	101	apple green	round	big	010
3	102	tomato	round	middle	100
4	103	orange	round	big	110
5	104	apricot	oval	middle	110
6	105	banana	long	big	110
7	106	cherry	round	small	100

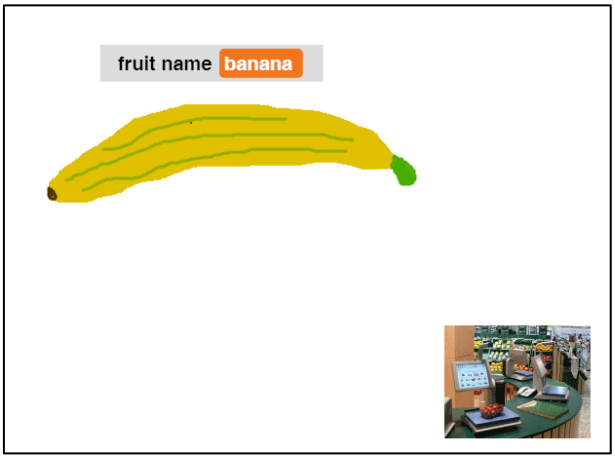




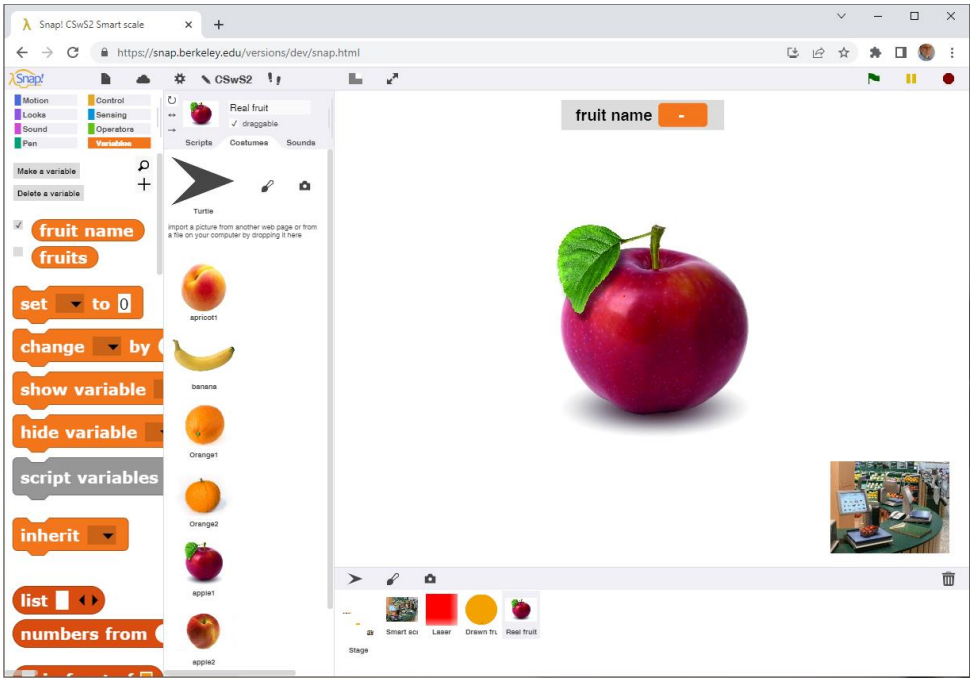
Und dieses Ergebnis können wir benutzen, um die Daten mit denen der gespeicherten Früchte zu vergleichen. Die sollen in einer Variablen *fruits* vorliegen, in der die Artikelnummer, die Bezeichnung und die typischen Fruchtdaten abgelegt sind.



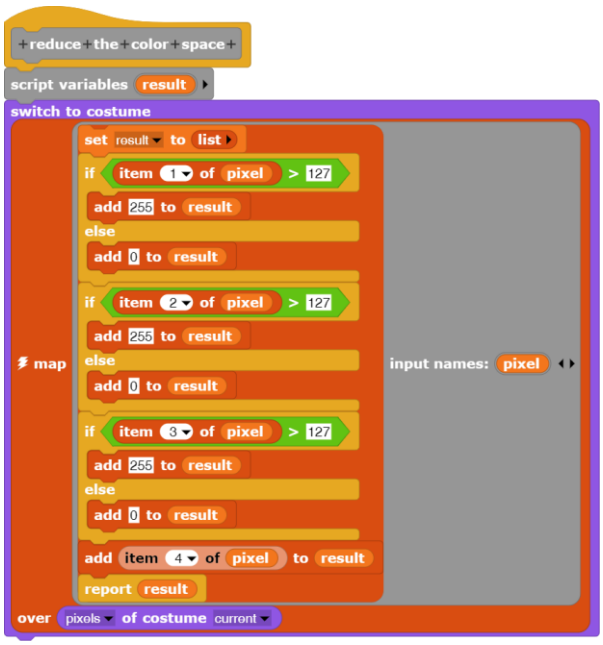
Klappt doch!



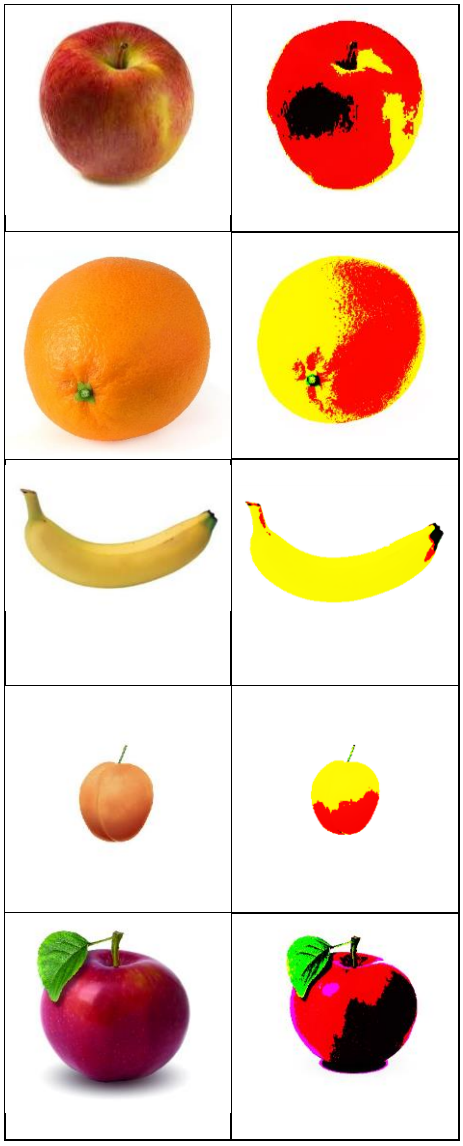
Nach diesen Erfolgen wird die Crew der Früchtewaage mutig und versucht es mit echten Früchtebildern.



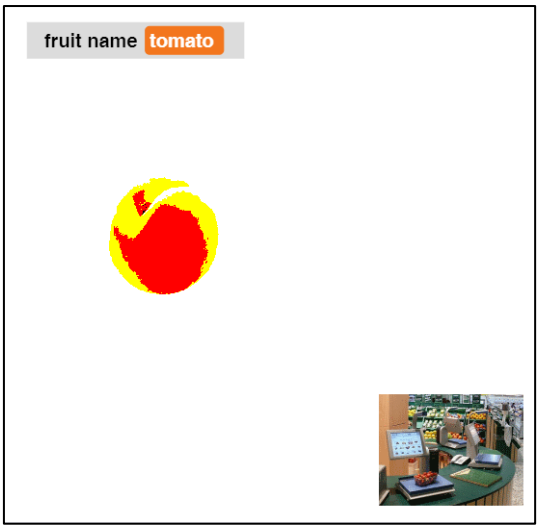
Auch deren Farbräume sollten reduziert werden, ähnlich wie bei den gezeichneten Früchten. Dann erhalten wir wieder ein farblich reduziertes Bild auf der Bühne.



Wir reduzieren die Farbenanzahl wie beschrieben …



… und stempeln das Bild auf die Bühne. Danach rufen wir die vorher entwickelte Früchtebestimmung erneut auf.



Na gut, an den Einträgen der Früchtetabelle sollten wir auch noch arbeiten. 😉

Wenn das geschehen ist, haben Sie den Werkzeugkasten zur optischen Früchtebestimmung zusammen:

1. Machen Sie ein Foto einer Frucht und wählen Sie es als Kostüm eines Sprites. Bilder kann man mit dem Smartphone oder der Laptopkamera machen. Der Hintergrund sollte weiß sein.
2. Reduzieren Sie die Farben des Bildes.
3. Messen Sie Größe und Form der Frucht.
4. Messen Sie die mittlere Farbe der Frucht und reduzieren Sie auch diese.
5. Berechnen Sie den Farbcode der Frucht.

Die erhaltenen Daten *Form*, *Größe* und *Farbcode* können als Spalten einer Datenbanktabelle benutzt werden. Wir haben dann je drei unterschiedliche Werte für Größe und Form sowie 8 mögliche Farbcodes. Damit können wir 3 * 3 * 8 = 72 Früchte unterscheiden. Probieren Sie mal eine „echte“ intelligente Früchtewaage im Kaufhaus aus – so schlecht sind wir gar nicht. 😉

Aufgaben:

1. Erzeugen Sie eine Datenbanktabelle der folgenden Art für Früchte:

pnr	Frucht	Form	Groesse	Farbcode
123	roter Apfel	rund	groß	100
223	Kirsche	rund	klein	100
456	Banane	lang	groß	110
…	…	…	…	…

2. Fügen Sie die Tabelle zu Ihrer Datenbank hinzu.

3. Schreiben Sie eine *auswerte*-Methode so, dass sie Bezeichnung und Preis der Frucht liefert. Benutzen Sie dafür Datenbankbefehle.

4. Das Farbreduzierungsverfahren ist sehr grob. Erfinden Sie ein besseres.

5. Unser Früchteerkennungsverfahren arbeitet nur gut, wenn die Frucht in der Bühnenmitte platziert und horizontal ausgerichtet ist. Wenn wir ein Sprite mit einem Früchtebild als Kostüm ausstatten, können wir das Sprite in der Mitte zentrieren und ausrichten, bevor wir das Kostüm drucken. Realisieren Sie das Verfahren.

6. Wenn wir einen detaillierteren Farbcode verwenden, können wir mehr Früchte unterscheiden. Wäre das in allen Situationen ein Fortschritt?

7. Es könnte ja sein, dass der Hintergrund der Frucht nicht weiß ist. Können Sie helfen?

17.7 KFZ-Kennzeichenerkennung

Altersstufe: *Sekundarstufe II* Material: *License plate recognition*

Der Erfolg mit der intelligenten Waage aus dem vorigen Projekt geht wie ein Lauffeuer durch den Supermarkt. Er erreicht auch die Sicherheitsabteilung. Die ist unter anderem für das Parkhaus verantwortlich. Um das Bezahlen der Parkgebühren zu vereinfachen, installiert die Abteilung eine automatische Kennzeichenerkennung. Registrierte Kunden mit Kundenkarte und automatischer Abrechnung müssen dann nicht mehr vor der Parkhausschranke halten – jedenfalls ist das die Hoffnung.

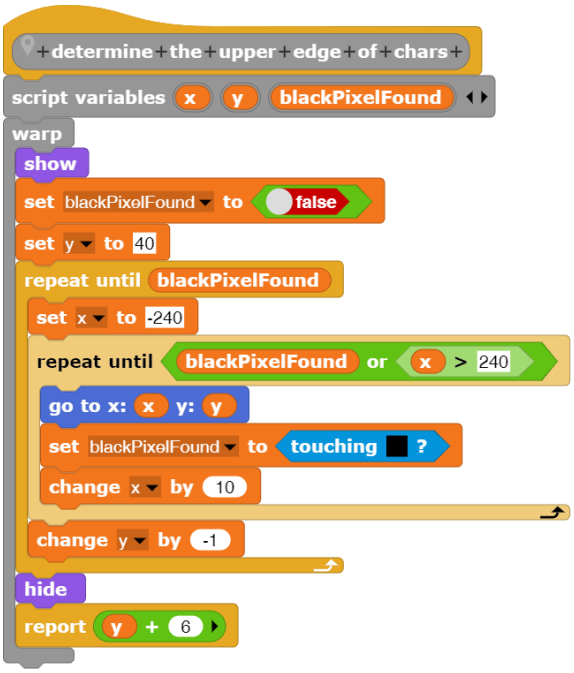
Autokennzeichen enthalten spezielle Zeichensätze, die die Zeichenerkennung durch Computer erleichtern. In Europa haben sie einen schwarzen Rand – und das ist gut für uns. Versuchen wir also, die Zahlen auf dem Kennzeichen zu bestimmen. (Die anderen Zeichen überlassen wir Ihnen.) Glücklicherweise haben wir schon fast alle Werkzeuge für unser Vorhaben realisiert. Sie müssen nur die Leute an der intelligente Früchtewaage fragen!

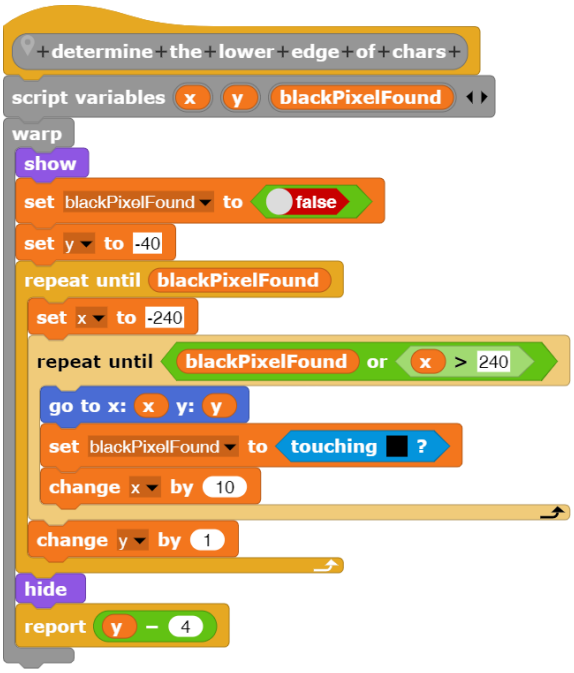
Wir versuchen, eine extrem einfache Methode zur Kennzeichenerkennung zu entwickeln. Das Ergebnis ist sehr empfindlich gegen Änderungen in Lage und Größe der Kennzeichen. Aber diese Nachteile können leicht korrigiert werden, indem ein detailliertes Messverfahren eingesetzt wird. Sehen Sie sich mal die Übungsaufgaben an!

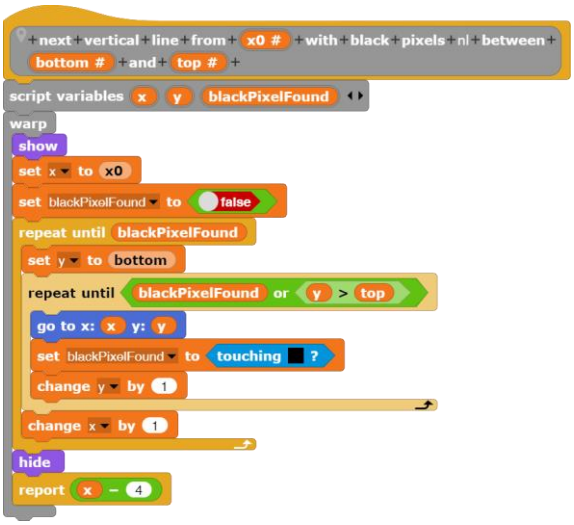
OCR (*Optical Character Recognition*) benutzt komplexe Methoden, oft mit Neuronalen Netzen, um Zeichen zu erkennen. Wir erfinden hier ein einfacheres Verfahren, das dem der intelligenten Waage ähnelt. Weil alle unsere Zeichen auf dem Nummernschild die gleiche Breite haben, können wir sie leicht identifizieren, wenn wir die Grenzen des Nummernschildes gefunden haben. Bei der intelligenten Waage können Sie sehen, wie das geschieht. Wir benutzen auch weiterhin deren Laser.

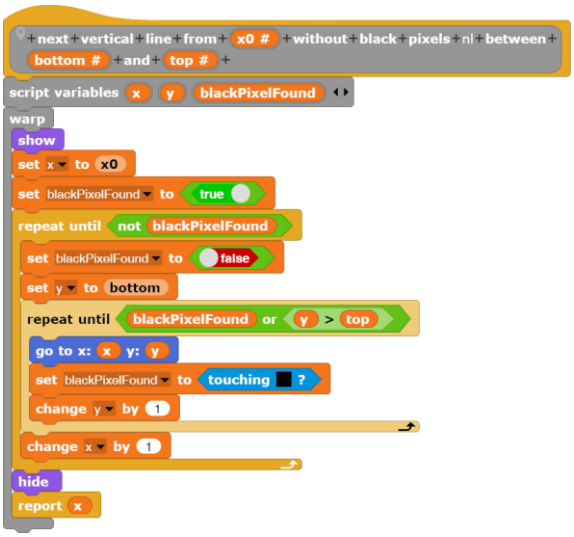
Nummernschilder können wir schnell mithilfe diverser Generatoren im Internet erzeugen. Wir speichern sie als Kostüme eines Sprites *Licence plate*. Nach dem Klicken auf die grüne Flagge stempeln wir das Kostüm auf die Bühne – wie bei der intelligenten Waage. Der relevante Bereich mit den Ziffern befindet sich dann zwischen $-240 < x < 240$ und $-40 < y < 40$.

Wir beginnen, indem wir oben und unten im Nummernschild Linien suchen, die keine schwarzen Pixel enthalten. Deren Positionen geben den oberen und unteren Rand der relevanten Zeichen an. Danach suchen wir von links nach rechts vertikale Linien, auf denen sich schwarze Pixel befinden. Wenn wir die erste gefunden haben, haben wir auch den Anfang des ersten Zeichens. Danach suchen wir nach der ersten vertikalen Linie ohne schwarze Pixel. Deren x-Position ergibt das Ende des ersten Zeichens. Wir haben ein „Fenster“ mit dem ersten Zeichen darin. Die nächste Linie mit schwarzen Pixeln ergibt die Breite der Lücke zwischen den Zeichen.

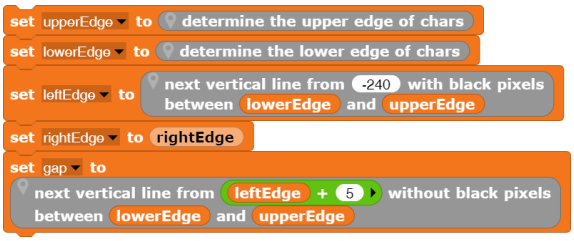




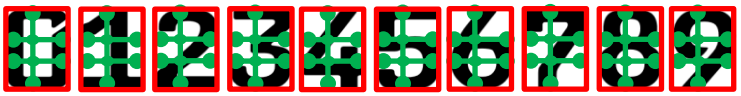




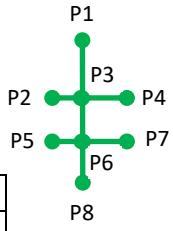
Mithilfe dieser Blöcke bestimmen wir die Grenzen der ersten Ziffer und den Abstand zur zweiten. Danach zeichnen wir das umschreibende Rechteck in Rot.



Dieses Fenster können wir jetzt gedanklich über alle Zeichen des Nummernschilds schieben und versuchen, die Zeichen innerhalb des Feldes zu erkennen.



Es fehlt noch die Ziffernerkennung selbst. Als Ausgangspunkt nehmen wir die Zeichen mit dem Rechteck rund herum. Wir stellen uns ein „Sensorfeld“ aus drei sich kreuzenden Linien vor. An den runden Punkten messen wir die Farben. Wir nummerieren die Punkte wie gezeigt und sehen uns die Resultate in Tabellenform an. (graue Felder: Ergebnis schwer vorherzusagen)



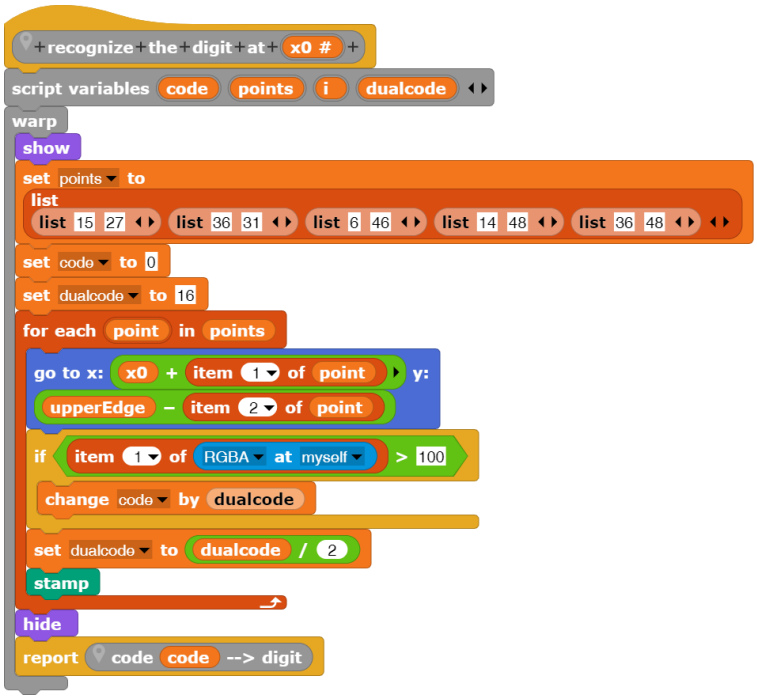
Zeichen	P1	P2	P3	P4	P5	P6	P7	P8	Code(s)
0	■	■		■	■		■	■	00100100
1	■							■	01111110
2	■			■		■		■	01101010
3	■		▒				■	■	01011100 01111100
4			■		■	■	■		11010001
5	■	■	■	■			■	■	00001100
6	■		■	■	■		■	■	0100100
7	■					■		■	01111010
8	■	▒	■		■		■	■	00010100 01010100
9	■	■		■			▒	■	00101100 00101110

Bei den Zeichen *3*, *8* und *9* können Fehler auftreten, wenn die Punkte nicht gut justiert sind. Aber das macht nichts, denn wenn wir die Sensoren *P2*, *P3* und *P7* etwas verschieben, sodass sie klare Werte liefern, können wir auf die Sensoren *P1*, *P2* und *P8* (z. B.) verzichten und haben trotzdem einen brauchbaren Code.

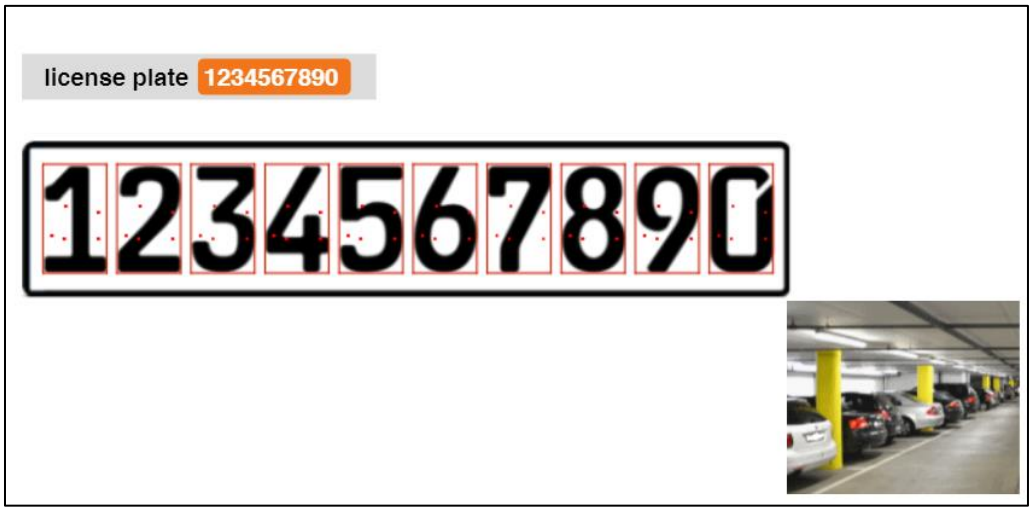
zeichen	P1	P2	P3	P4	P5	P6	P7	P8	Code	Wert
0	■	■		■	■		■	■	10010	18
1	■	▒						■	11111	31
2	■	▒		■		■		■	10101	21
3	■	▒					■	■	11110	30
4	▒	▒	■		■	■	■	▒	01000	8
5	■	■		■			■	■	10110	22
6	■	▒	■	■			■	■	00010	2
7	■	▒				■		■	11101	29
8	■	▒	■		■		■	■	01010	10
9	■	■		■				■	10111	23

Ein mögliches Layout für die restlichen Sensoren wäre:

Wir wählen ein Nummernschild mit allen zehn Zeichen. Die Sensoren werden an geeigneten Stellen platziert (hier: (15|27), …) und in einer Liste gespeichert, um an den Positionen die Farben im Zeichenfenster zu lesen und eine Codenummer aus den Farben, interpretiert als dualer Code, zu bilden. Danach transformieren wir den Code zum richtigen Zeichen.



Jetzt kann die Sicherheitsabteilung von ihrem Büro im Parkhaus aus den Laser fragen, welches Auto gerade gekommen ist:



Das Ergebnis beeindruckt besonders die Werbeabteilung des Supermarkts, die sofort ganz neue Anwendungen des Verfahrens sieht. Alle sind sehr stolz auf die Sicherheitsleute!

Aufgaben

1. Die Zeichenerkennung in den Beispielen ist sehr einfach, aber sehr empfindlich gegen Änderungen in Größe und Position des Nummernschildes. Benutzen Sie mehr Sensoren, um die Zeichen sicherer zu erkennen.

2. Erweitern Sie die Zeichenerkennung auf den gesamten Zeichensatz für KFZ-Kennzeichen.

3. Zeichenerkennungsprogramme können lernen. Falls das Skript keine erkennbaren Muster findet, sollte es sein Ergebnis anzeigen und nach dem richtigen Zeichen fragen. Speichern Sie die Muster und die dazu gehörigen Zeichen in einer Datenbank-Tabelle. Benutzen Sie Abfragen, um unbekannte Muster zu identifizieren.

4. Wenn man schmutzige Nummernschilder lesen will, findet man keine scharfen Zeichengrenzen. Als Folge werden einige Sensoren Fehler produzieren. Verbessern Sie die Resultate in solchen Fällen, indem Sie den „nächsten richtigen Code“ eines falschen Codes bestimmen.

5. Die Erkennung von verschmutzten Kennzeichen kann verbessert werden, wenn das Farbbild in ein reines Schwarz-Weiß-Bild umgewandelt wird und die durch den Schmutzt entstandenen Lücken geschlossen werden. Informieren Sie sich über für diesen Zweck geeignete Verfahren und realisieren Sie eins davon.

6. Die Sicherheitsabteilung benötigt eine Datenbank mit Nummernschildern und Fahrzeugbesitzern sowie deren Status (Kunde, Betriebsmitglied, unerwünschte Person, externer Parker, …). Können Sie helfen?

7. Die Nummernschilderkennung stellt sich als großer Erfolg der Sicherheitsabteilung heraus. Alle ihre Mitglieder sind sehr stolz darauf und die anderen Betriebsmitglieder bewundern die „Sheriffs“. Die Werbeabteilung möchte nun die Daten der Nummernschildtabelle nutzen, um häufig und lange im Supermarkt anwesende Kunden als VIP-Kunden zu ehren. Diese erhalten spezielle Parkplätze in der Nähe des Fahrstuhls. Schreiben Sie eine Anfrage, um VIP-Kunden zu finden.

8. Nach einiger Zeit sind die VIP-Parkplätze mit den Autos von Rentnern und Arbeitslosen besetzt. Deshalb erweitert die Werbeabteilung die Kriterien für VIP-Kunden um ein Minimum an Umsatz bei deren Einkäufen. Weil fast alle Kunden Kreditkarten beim Bezahlen benutzen, ist das auch kein Problem. Verbessern Sie die VIP-Kunden-Abfrage entsprechend.

9. Die Werbeabteilung stellt fest, dass es hilfreich wäre, nicht nur den Umsatz eines Kunden zu kennen, sondern auch, was er gekauft hat. Wenn sie die Interessen der Kunden kennt, kann sie diese mit speziellen Angeboten und Sonderpreisen versorgen. Bestimmen Sie die dafür notwendigen zusätzlichen Tabellen und deren Spalten in der Datenbank. Schreiben Sie geeignete Anfragen.

10. Die Werbeabteilung möchte wissen, ob ihre Werbemaßnahmen erfolgreich sind. Erreicht sie die Kunden? Versuchen Sie diese Fragen auf der Grundlage der gespeicherten Daten zu beantworten.

Wie kann man …