Programmieren mit

Michael Brenner

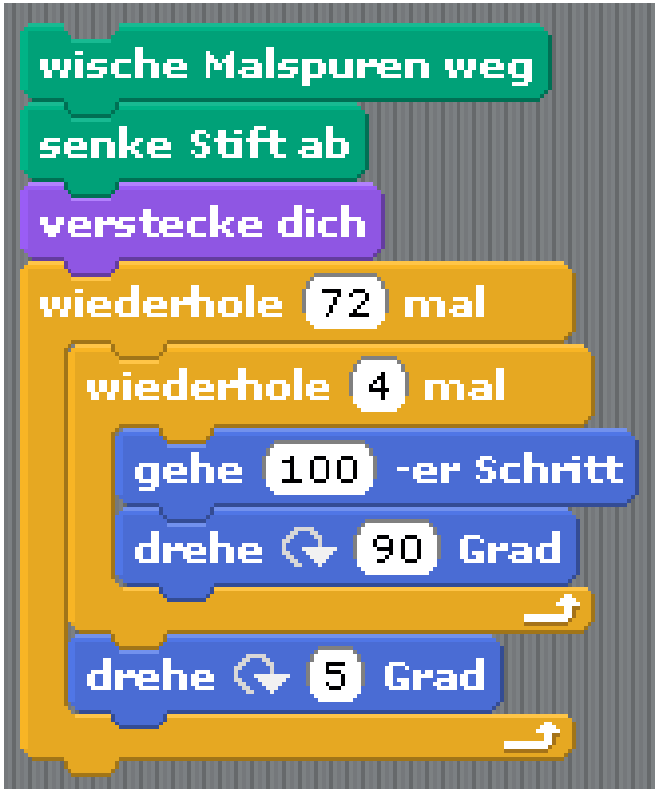
Freiburg im Breisgau

Der heutige Workshop

„In zwei Stunden von der Grundschule ins Informatikstudium!“

- Einzelbefehle → Kontrollstrukturen -→ Unterprogramme → Parameter
- Variablen
- Rekursion
- Listen und Lambdas
- Quicksort für Eilige
- Was tun, wenn‘s keine *while*-Schleife gibt?

SCRATCH

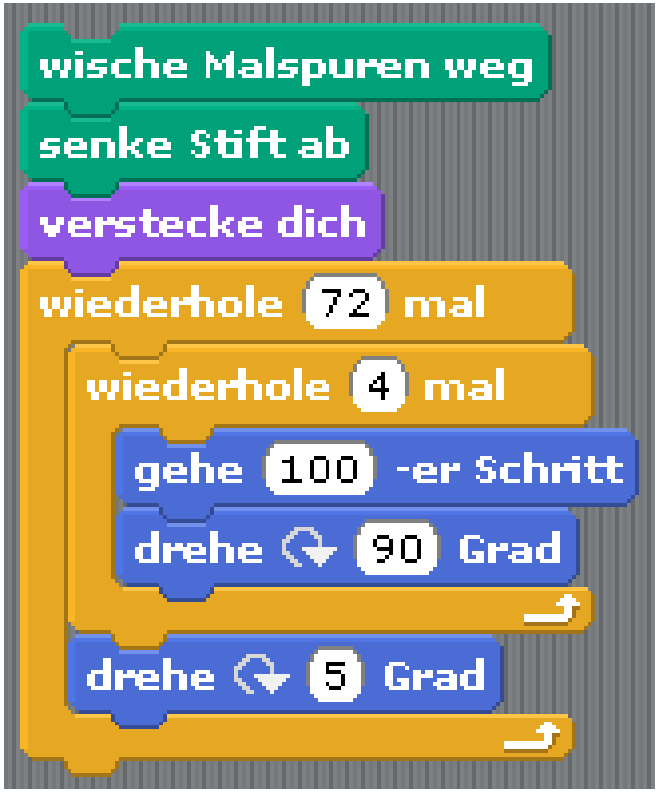


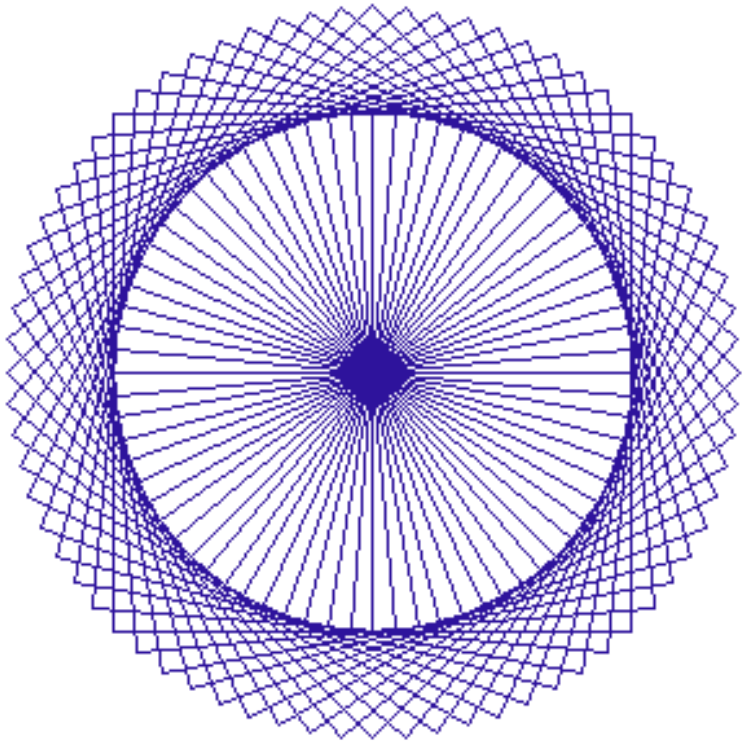
- entwickelt seit 2003 am MIT
- inspiriert von LOGO
- visuelle Programmiersprache
  - Programm = Puzzle aus Blöcken
  - „ausführbares Flussdiagramm“
  - kein Tippen – keine Syntaxfehler

→Ideal für Programmieranfänger, v.a. Kinder und Jugendliche


forever
imagine
program
share

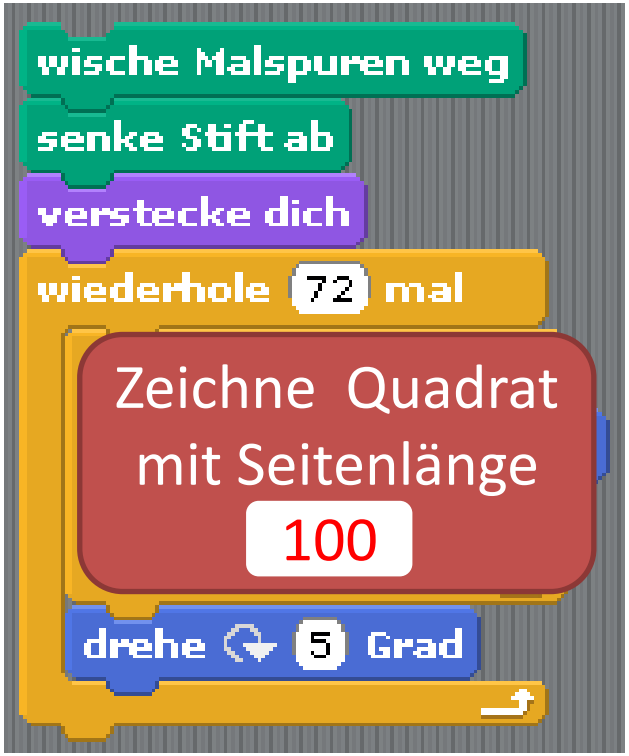
SCRATCH


wische Malspuren weg
senke Stift ab
verstecke dich
wiederhole 72 mal
wiederhole 4 mal
gehe 100 -er Schritt
drehe 90 Grad
drehe 5 Grad



Von SCRATCH zu Snap!

anschaulicher wäre doch…



In Scratch können **keine eigenen Blöcke** (d.h. Prozeduren, Funktionen, Unterprogramme) definiert werden !!!

- Brian Harvey (UC Berkeley) und Jens Mönig (deutscher Jurist (!) und Softwareentwickler)
- Vorläufer: BYOB (Build Your Own Blocks)

→Eigene Blöcke (Funktionen) definieren

- First-Class functions (Blöcke als Variableninhalte, Funktionsparameter und –ergebnisse)
- prozedurale, objektorientierte, funktionale Programmierung möglich
- *„Snap! is Scheme disguised as Scratch.“*

Programmieren im Browser

- Snap muss nicht installiert werden, sondern
- läuft als JavaScript-Anwendung im Browser.
- Programme speichern
  - in der Snap-Cloud oder
  - als XML-Datei lokal

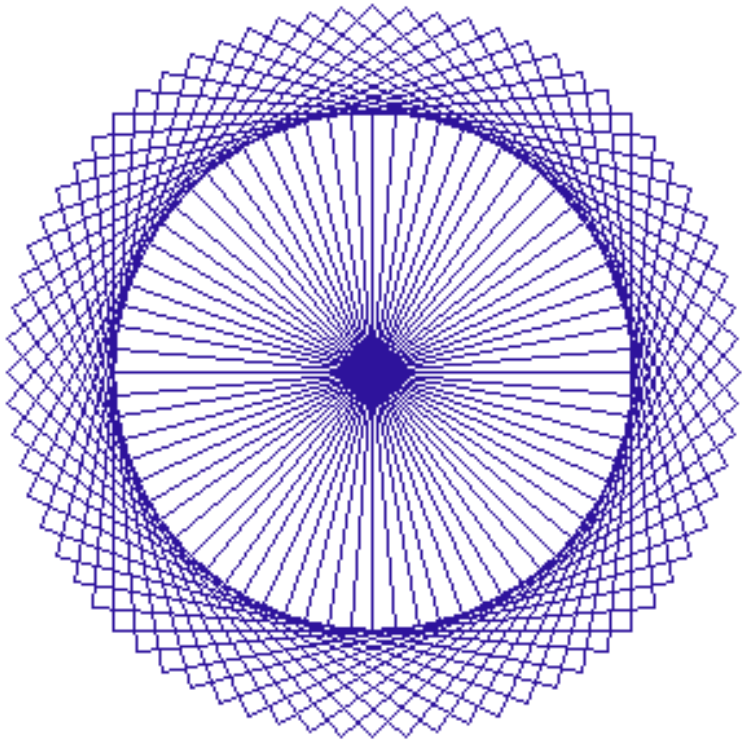
http://snap.berkeley.edu/run

oder

Websuche nach: „run snap“

Aufgabe: 72 Quadrate

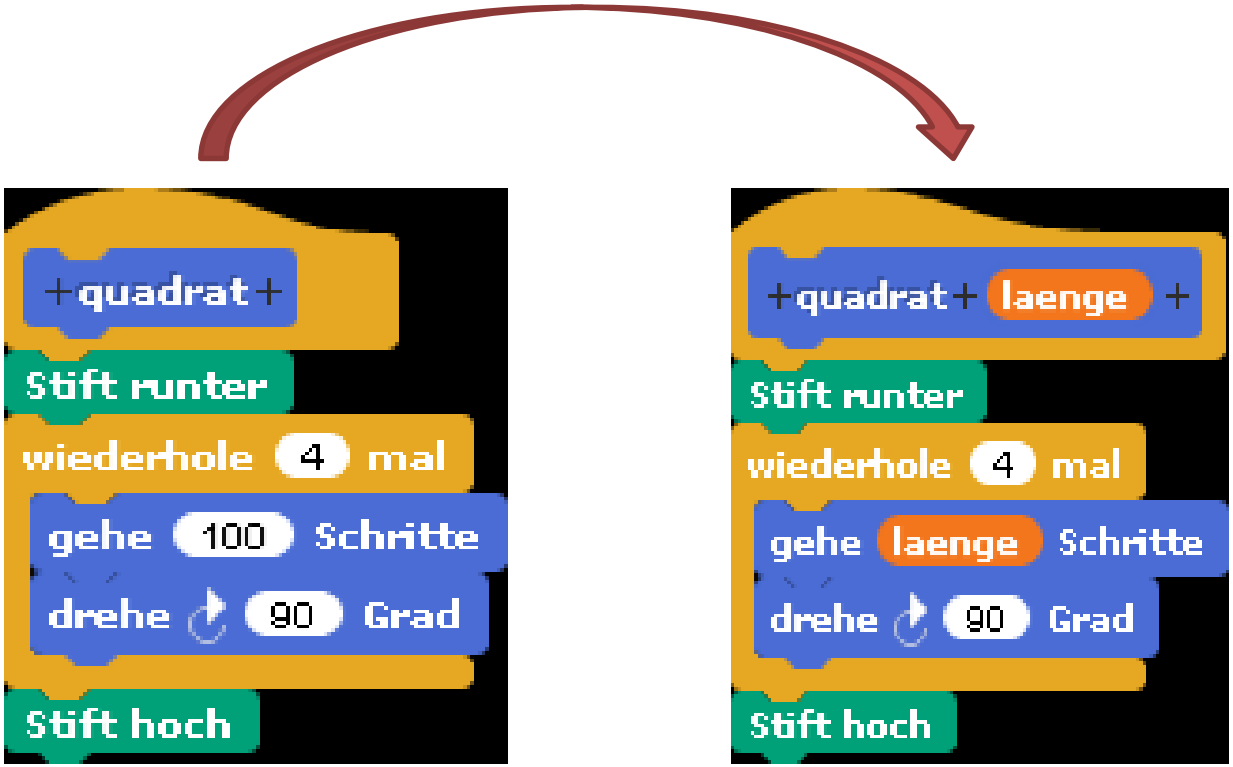
- Erstellen Sie einen neuen Block **Quadrat**, der ein Quadrat der Seitenlänge 100 zeichnet.
- Verwenden Sie **Quadrat**, um die nebenstehende Grafik zu erzeugen
- Experimentieren Sie mit Farben, Strichstärken, Winkeln, …



Freiheit durch Parameter:

Von 

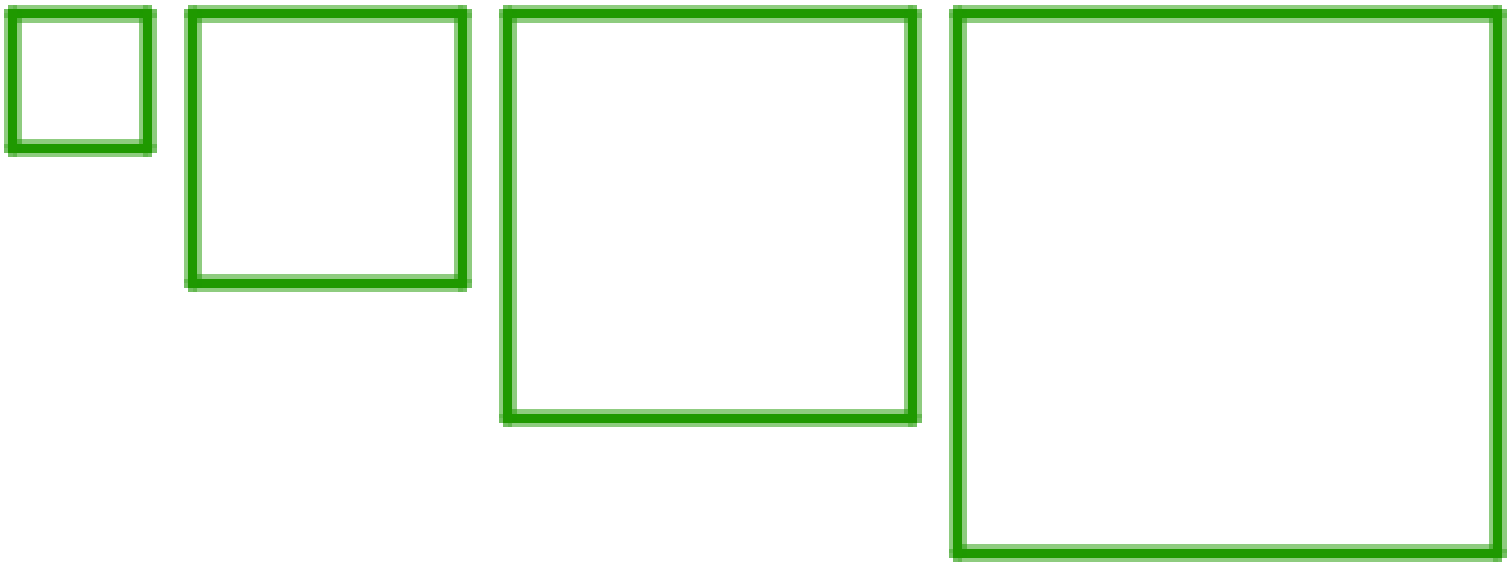
zu 



Freiheit durch Parameter:

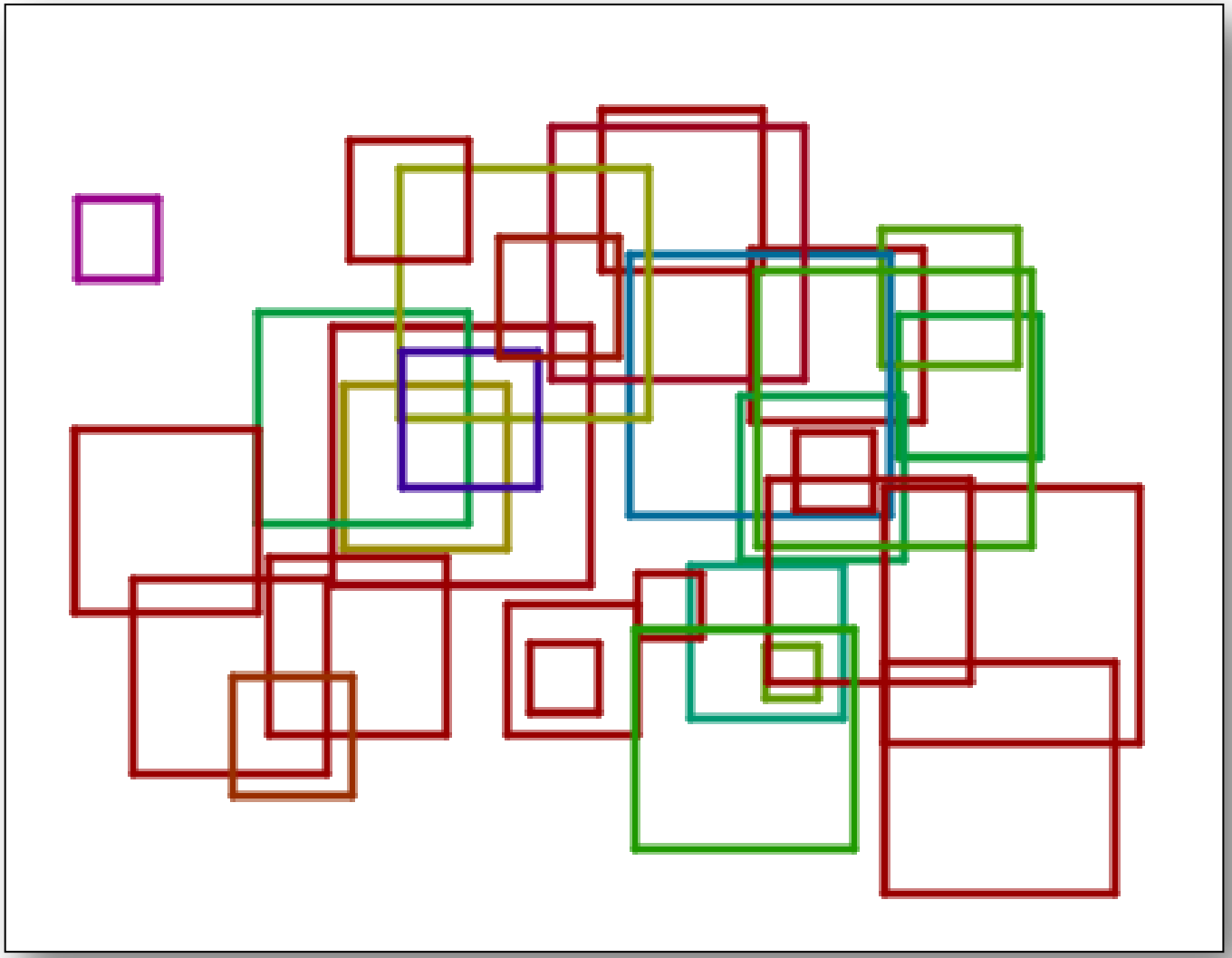
Von 

zu 



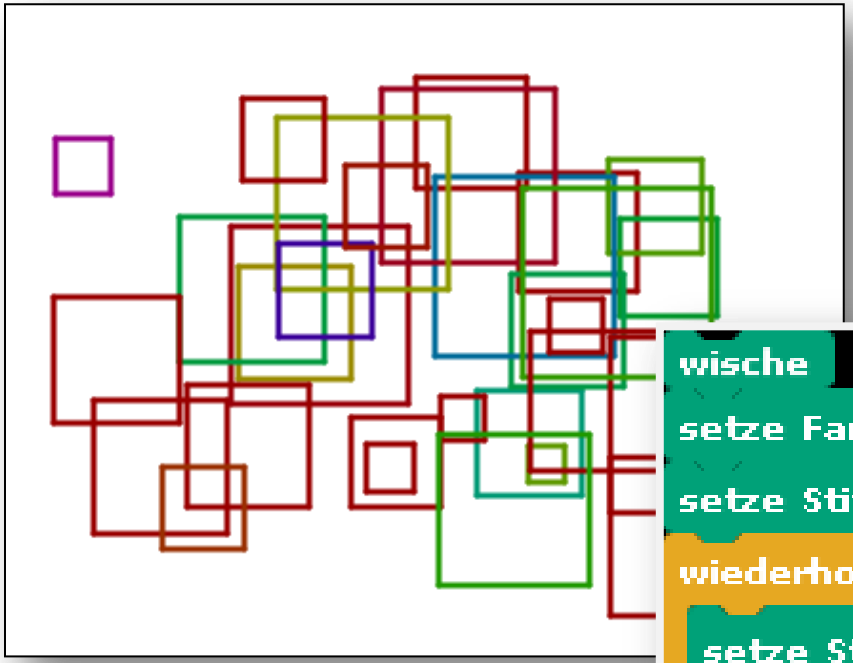
Freiheit durch Parameter:

Von quadrat zu quadrat 100



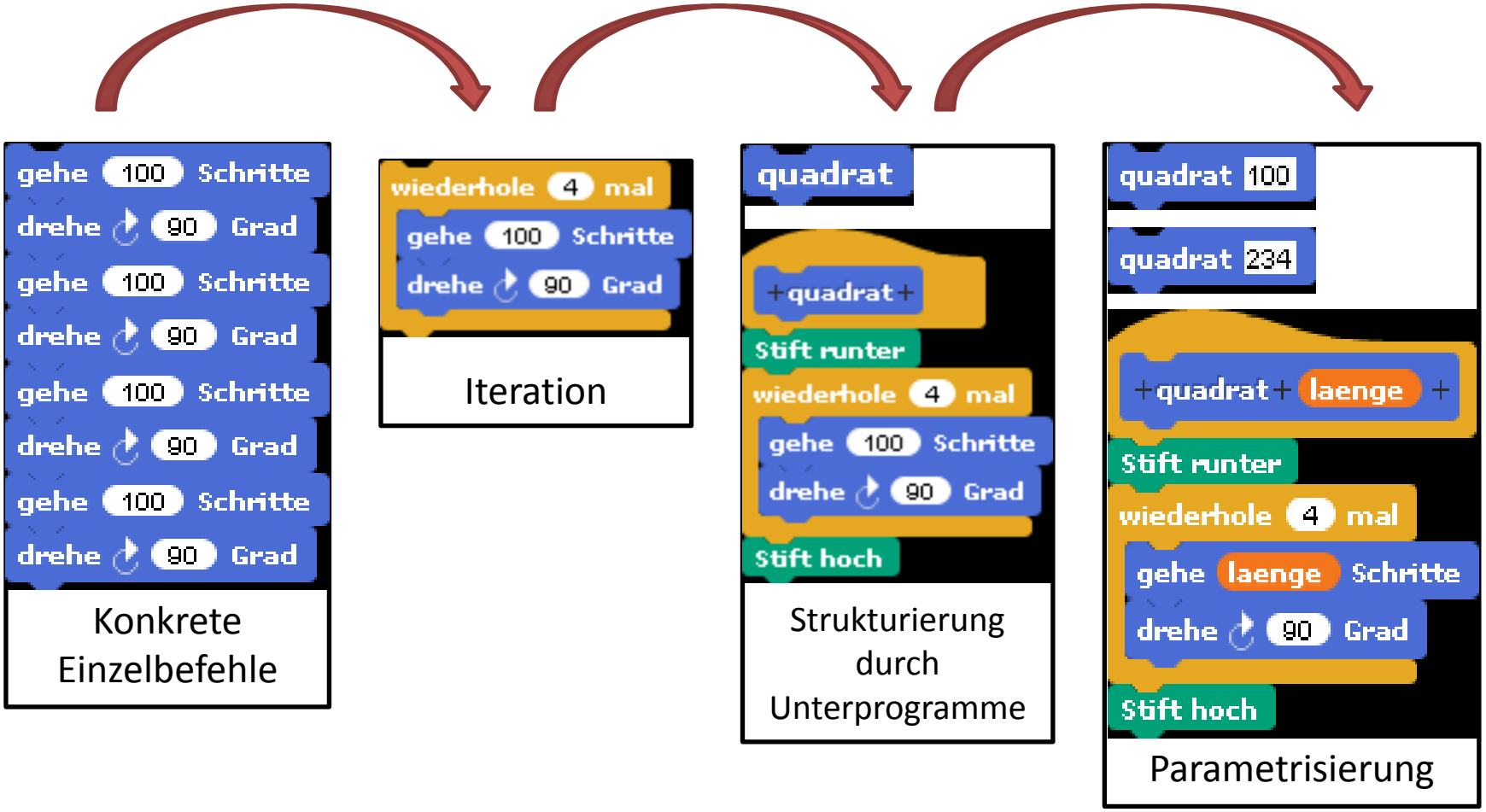
Freiheit durch Parameter:

Von quadrat zu quadrat 100



```
wische
setze Farbstärke auf 60
setze Stiftdicke auf 3
wiederhole 30 mal
    setze Stiftfarbe auf Zufallszahl von 0 bis 170
    gehe zu x: Zufallszahl von -200 bis 100 y:
    Zufallszahl von -70 bis 150
    quadrat Zufallszahl von 10 bis 100
```

Lernziel: Abstraktion & Generalisierung



Links zu den Programmbeispielen

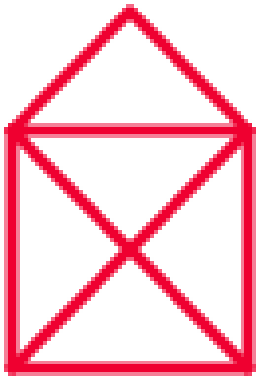
- Quadrat wie in Scratch1: http://is.gd/quadrat0
- Quadrat als eigener Block, aber mit fester Länge: http://is.gd/quadrat1
- Quadrat parametrisiert: http://is.gd/quadrat2Parameter
- Parametrisiertes Quadrat + Zufall: http://is.gd/quadrat3ParameterRandomisiert

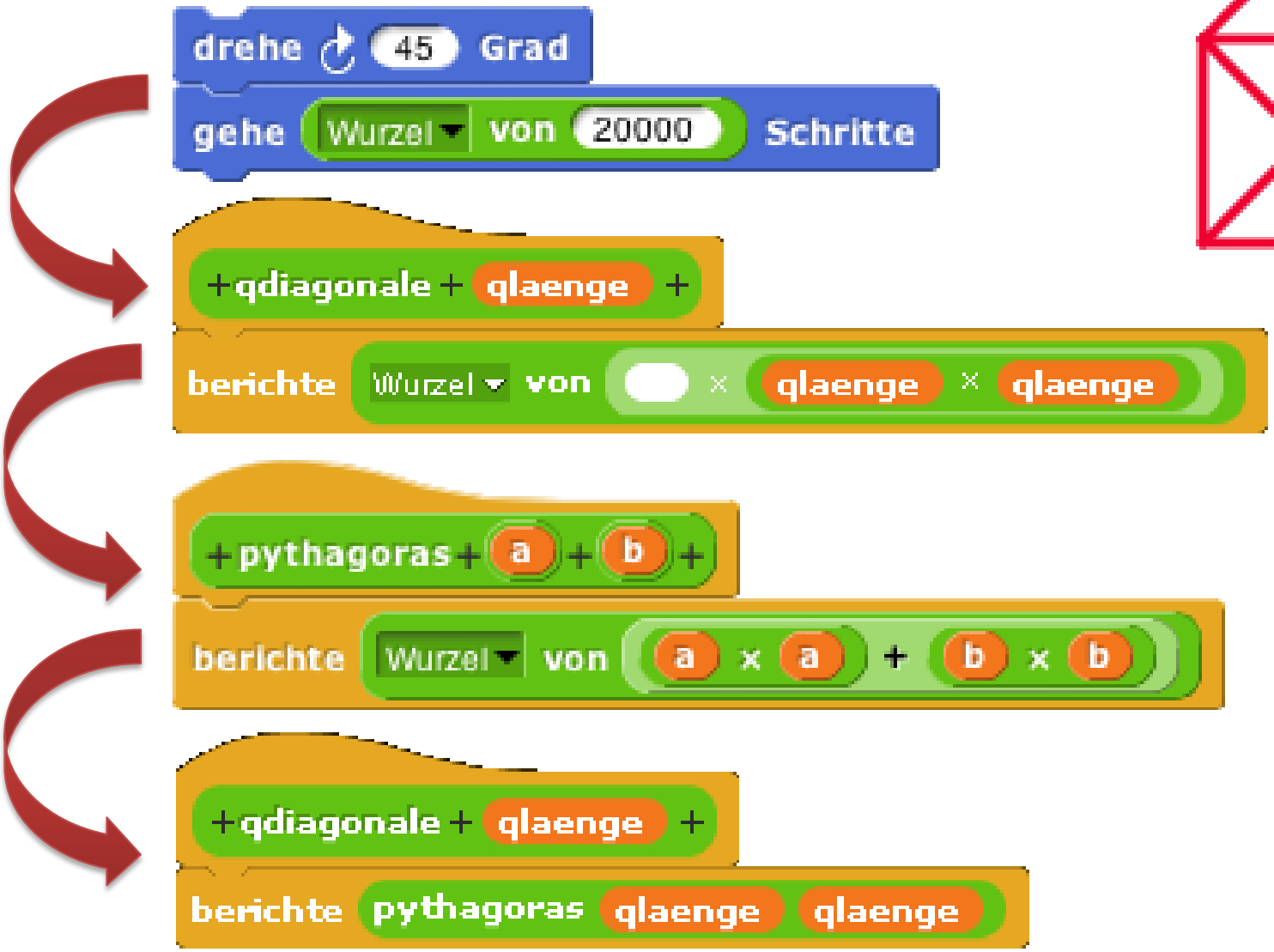
Das ist das Haus vom …

Tipps

- Tipp 1: Sie dürfen gerne Dinge verwenden, die Sie schon vorher programmiert haben.
- Tipp 2: Ja, bei den Diagonalen muss man etwas rechnen. Aber bitte nicht im Kopf – Sie sitzen doch am Computer! Nutzen Sie die grünen Blöcke im Bereich **Operatoren**.
- Tipp 3: Sie können, wenn Sie wollen, sogar selbst solch einen grünen Block, d.h. eine mathematische Funktion, erzeugen. Finden Sie selbst heraus, wie?
- Tipp 4: Der return-Befehl aus anderen Programmiersprachen heißt in Snap übrigens **report** bzw. **berichte** (im Bereich **Steuerung**).

Lernziel: Abstraktion & Generalisierung

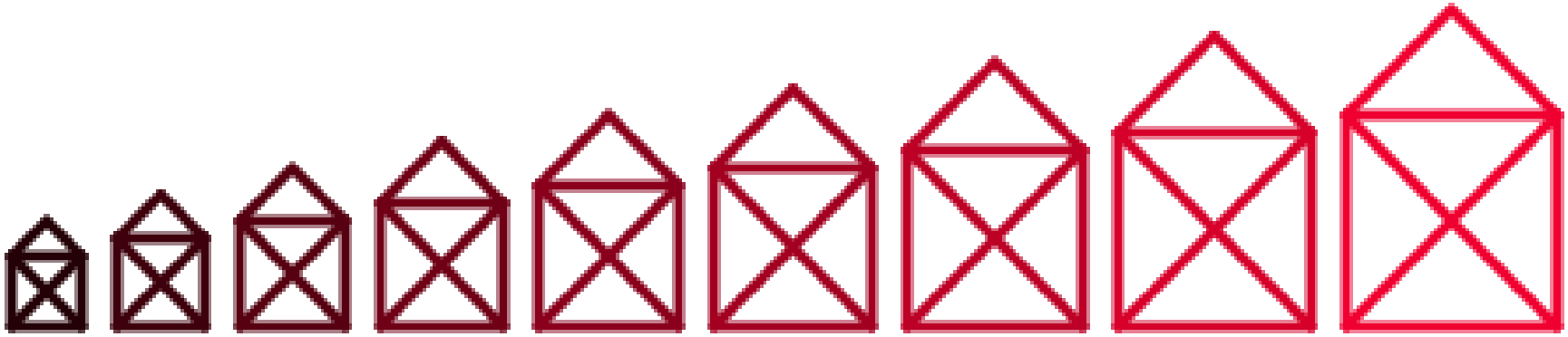




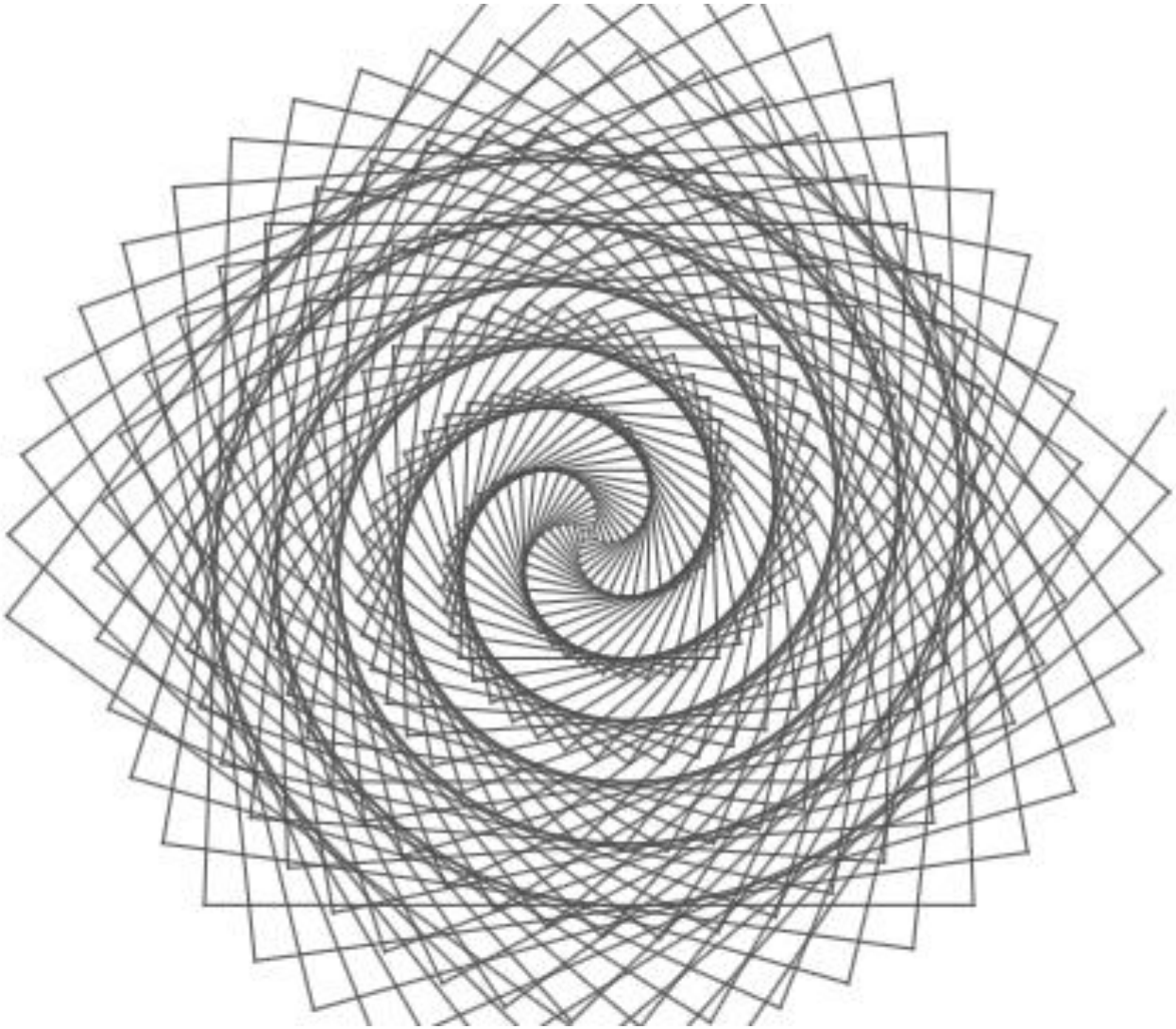
Links zu den Programmbeispielen

- Nikolaus mit fixer Schrittlänge, ohne Hilfsfunktion: http://is.gd/nikolaus0Fix
- Nikolaus mit Hilfsfunktion: http://is.gd/nikolaus1DiagonaleAlsFunktion
- Verallgemeinerung (Pythagoras) und Neudefinition der Hilfsfunktion: http://is.gd/nikolaus2Pythagoras

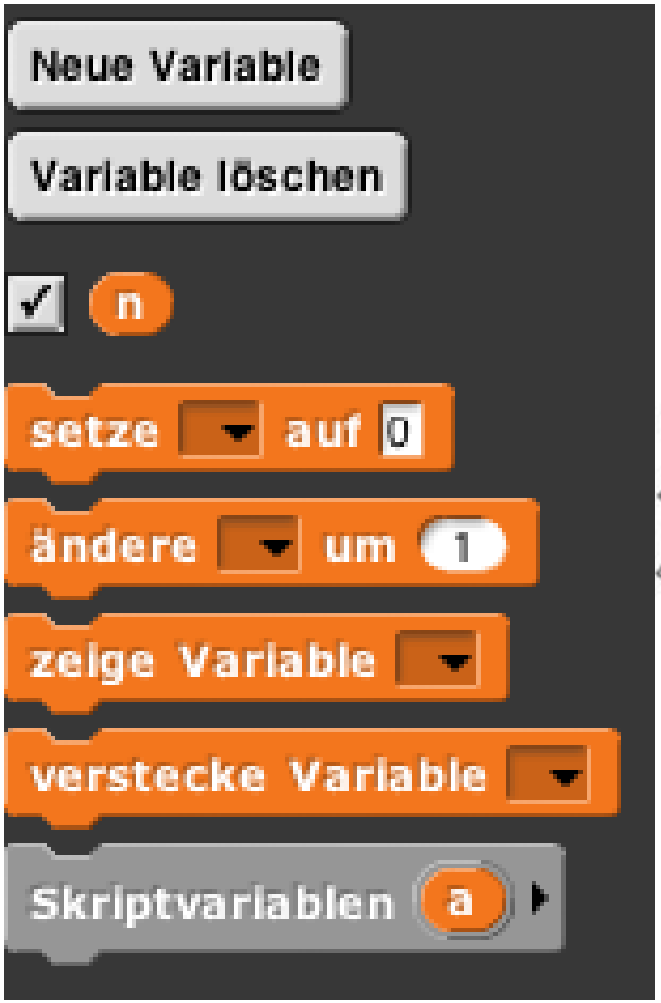
Und wie geht das?

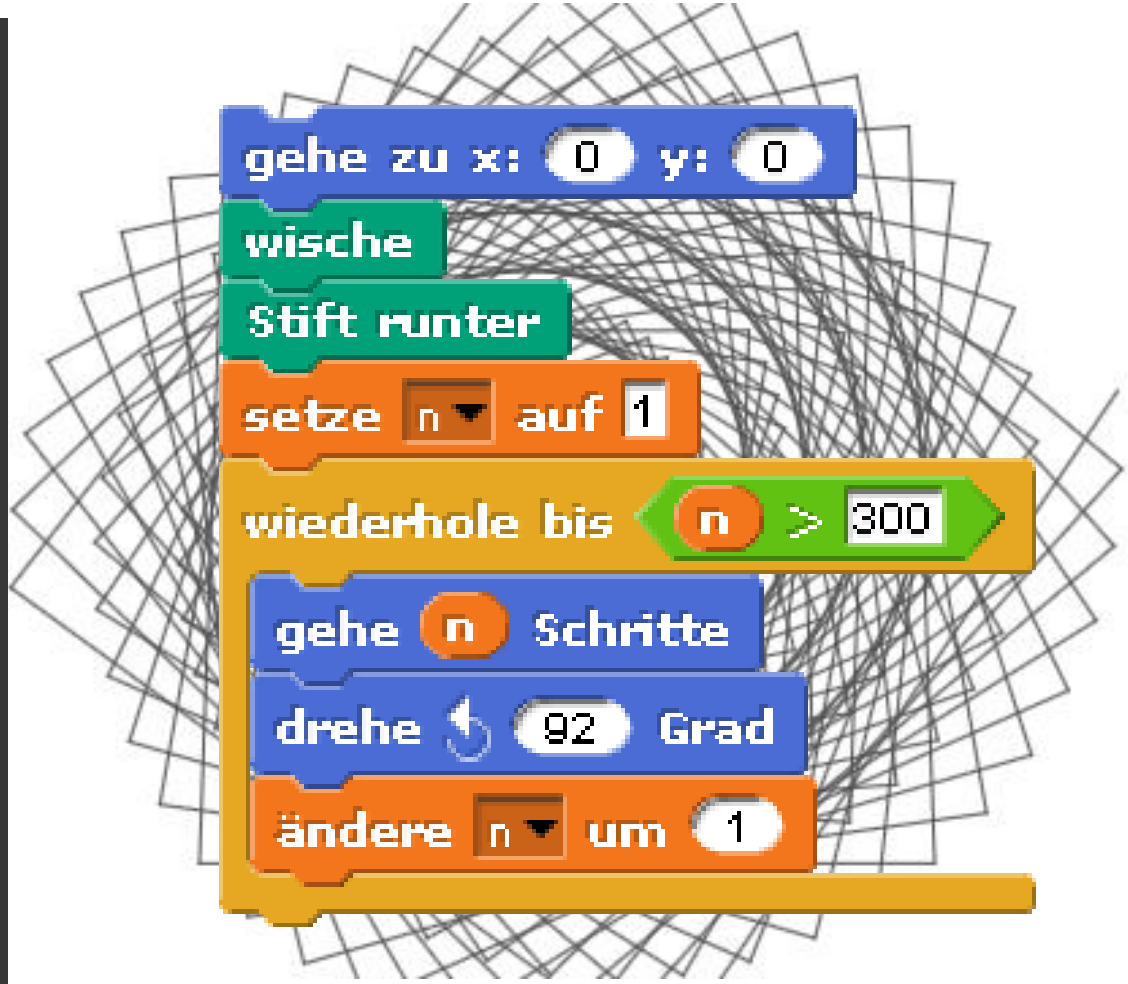


Oder das?



Variablen

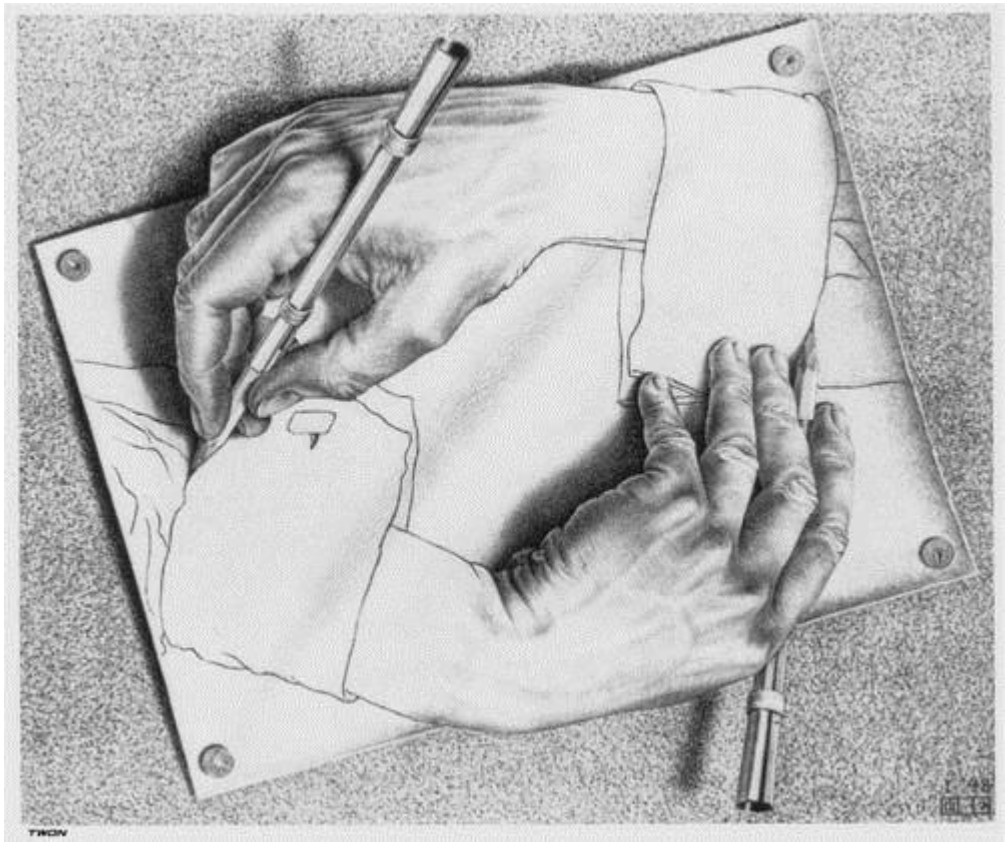

Neue Variable
Variable löschen
n
setze auf 0
ändere um 1
zeige Variable
verstecke Variable
Skriptvariablen a


gehe zu x: 0 y: 0
wische
Stift runter
setze n auf 1
wiederhole bis n > 300
gehe n Schritte
drehe 92 Grad
ändere n um 1

Links zu den Programmbeispielen

- Dorf der Nikoläuse: http://is.gd/nikolaus3Variable
- Spirale mit wachsender Schrittlänge: http://is.gd/Variablen1

Rekursion



Rekursion vermitteln – aber wie?

- Fakultät?

→ Laaaangweilig!

- Fibonacci?

→ Schlechtes Beispiel: Naive doppelt rekursive Implementierung hat ***exponentielle*** Laufzeit!

- Haus vom Nikolaus?

→ Eine Schleife tut‘s doch auch!

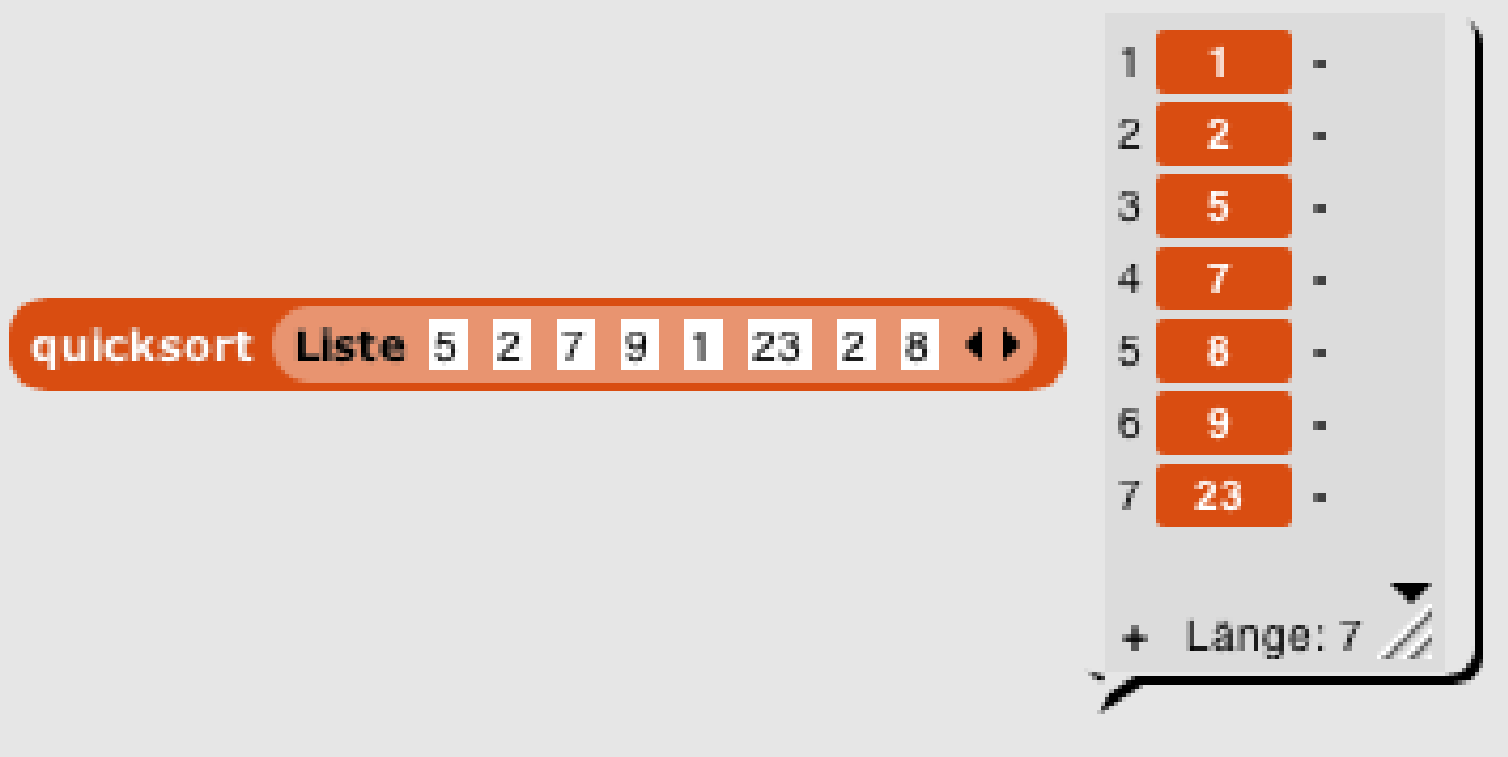
Links zu den Programmbeispielen

- Baum mit starren Winkeln:
  http://is.gd/Baum1
- Ein bisschen Zufall:
  http://is.gd/Baum2Randomisiert
- Experimentieren Sie mit weiteren Varianten!

Eine Animation sagt mehr als tausend Bilder

- http://is.gd/AnimationArmRekursiv
- Maus auf der „Bühne“ (im Ausführungsfenster) bewegen
- (Falls sich nichts tut, grüne Flagge anklicken.)
- Code anschauen und nach Belieben variieren!

Listen und Lambdas

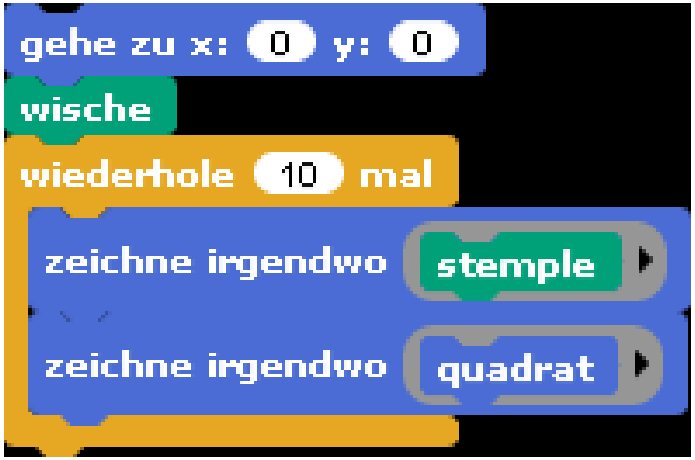


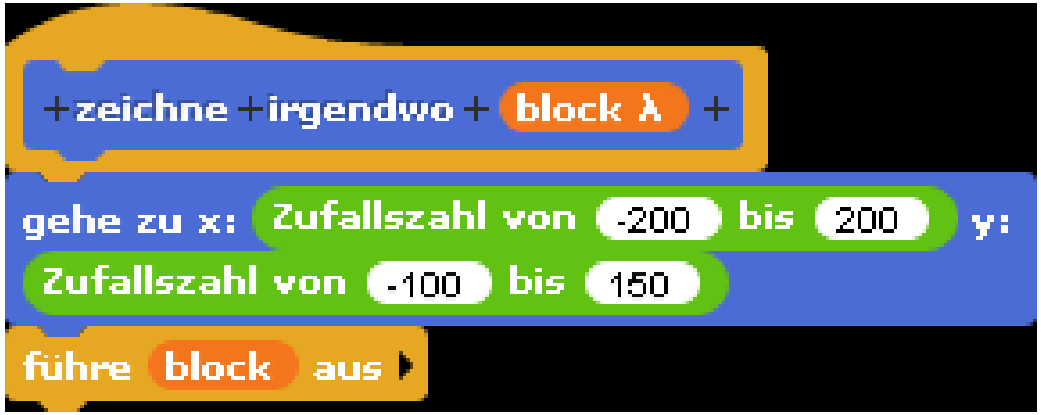
Listen

- Listen sind (im Gegensatz zu Scratch) *first class*:
  - können in Variablen gespeichert
  - an Funktionen übergeben
  - von Funktionen zurückgegeben werden
  - können verschachtelt werden
- Komplexe Datenstrukturen mit Listen implementierbar
  - Stack (Stapel)
  - Queue (Schlange)
  - Baum

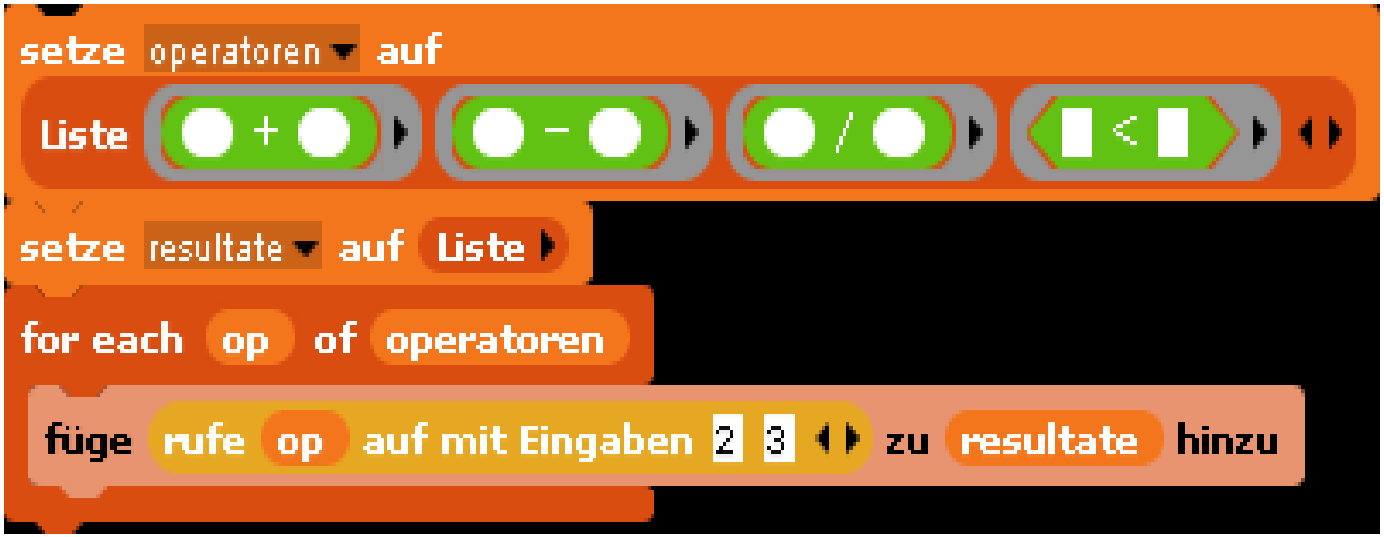
Higher-Order Funktionen

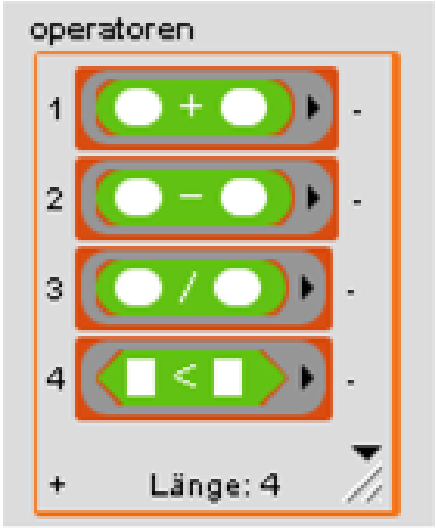
- Funktionen können Parameter und/oder Rückgabewert anderer Funktionen sein!

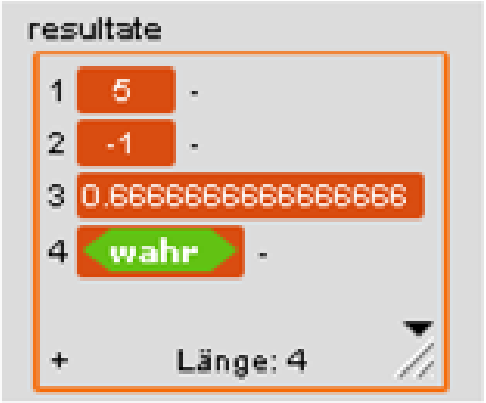




Listen voller Funktionen

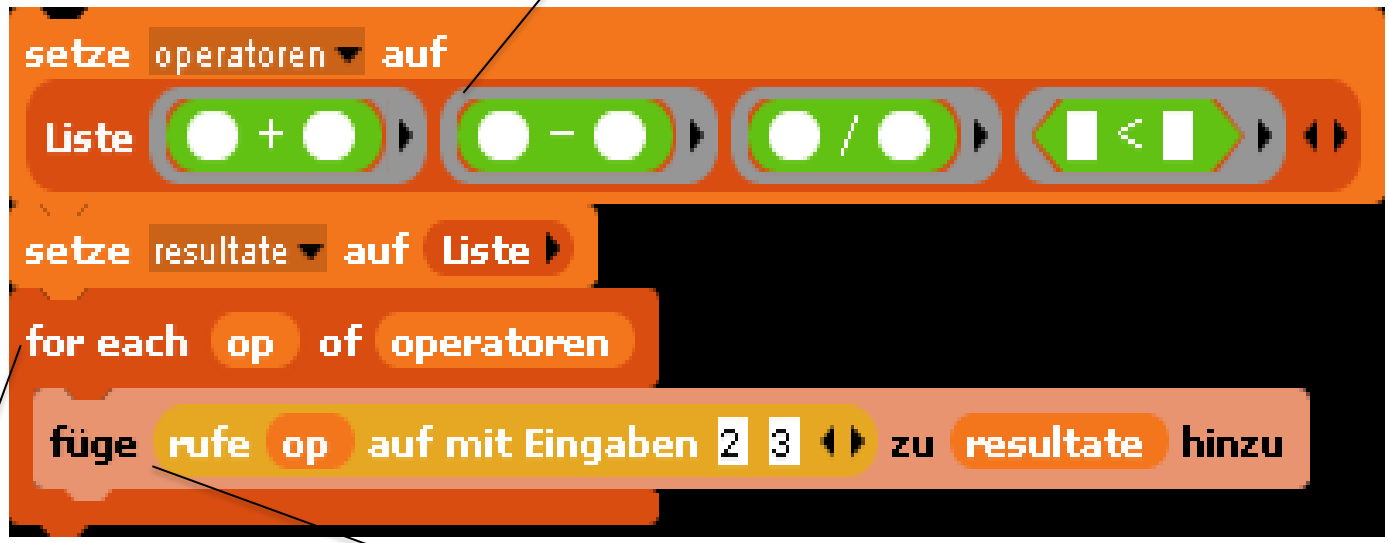






Listen voller Funktionen

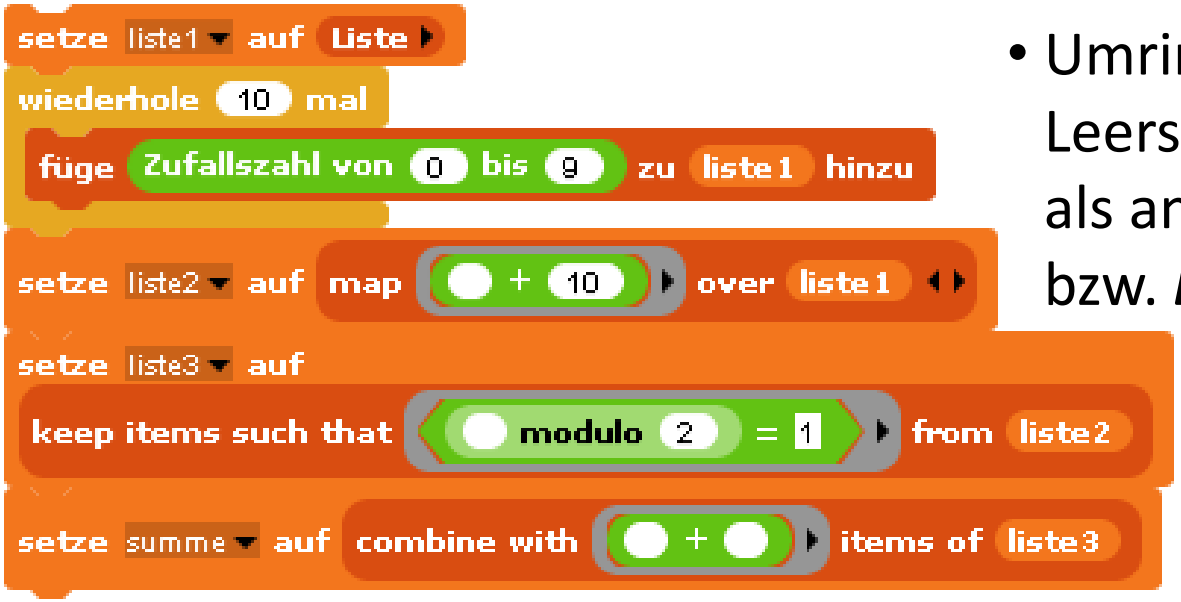
„Umringen“: Block hier noch nicht evaluieren



„for each“-Schleife aus der Tools-Bibliothek *(Datei → Tools laden)*

Jetzt wird der Block evaluiert, d.h. Funktion *op* aus der Liste wird mit Argumenten 2 und 3 aufgerufen.

Listenmanipulation funktional: *map*, *filter* und *reduce* à la Snap!



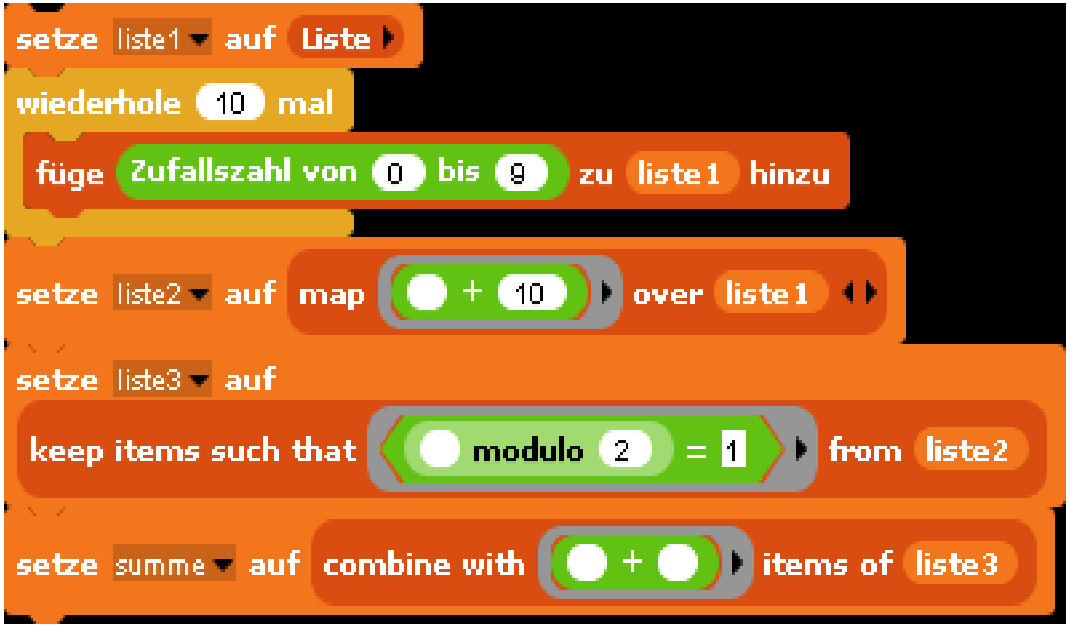
- Umringte Blöcke mit Leerstellen (hier z.B.: □+10) als anonyme Funktionen bzw. ***Lambda-Ausdrücke***

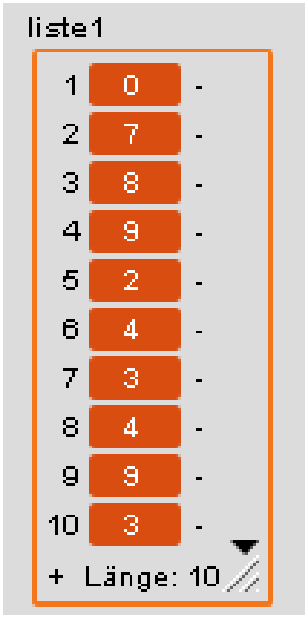
Klassische Funktionen zur Listenverarbeitung in *Tools*-Bibliothek:

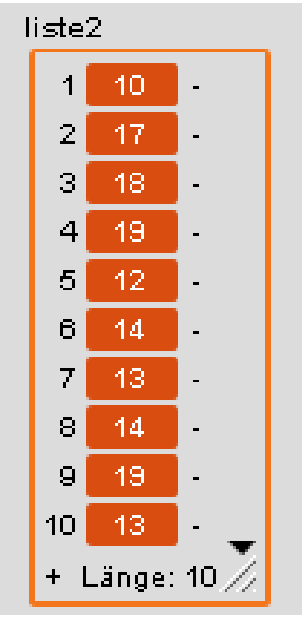
- *map*: Wende Funktion auf alle Elemente einer Liste an
- *keep items such that* (= *filter*): Behalte nur die Elemente, für die der Lambda-Ausdruck *wahr* ergibt
- *combine with* (= *reduce/fold*): „Kombiniere assoziative Funktionsaufrufe“

http://en.wikipedia.org/wiki/Fold_%28higher-order_function%29 ;-)

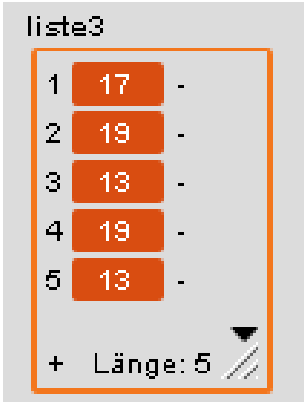
Listenmanipulation funktional: *map*, *filter* und *reduce* à la Snap!



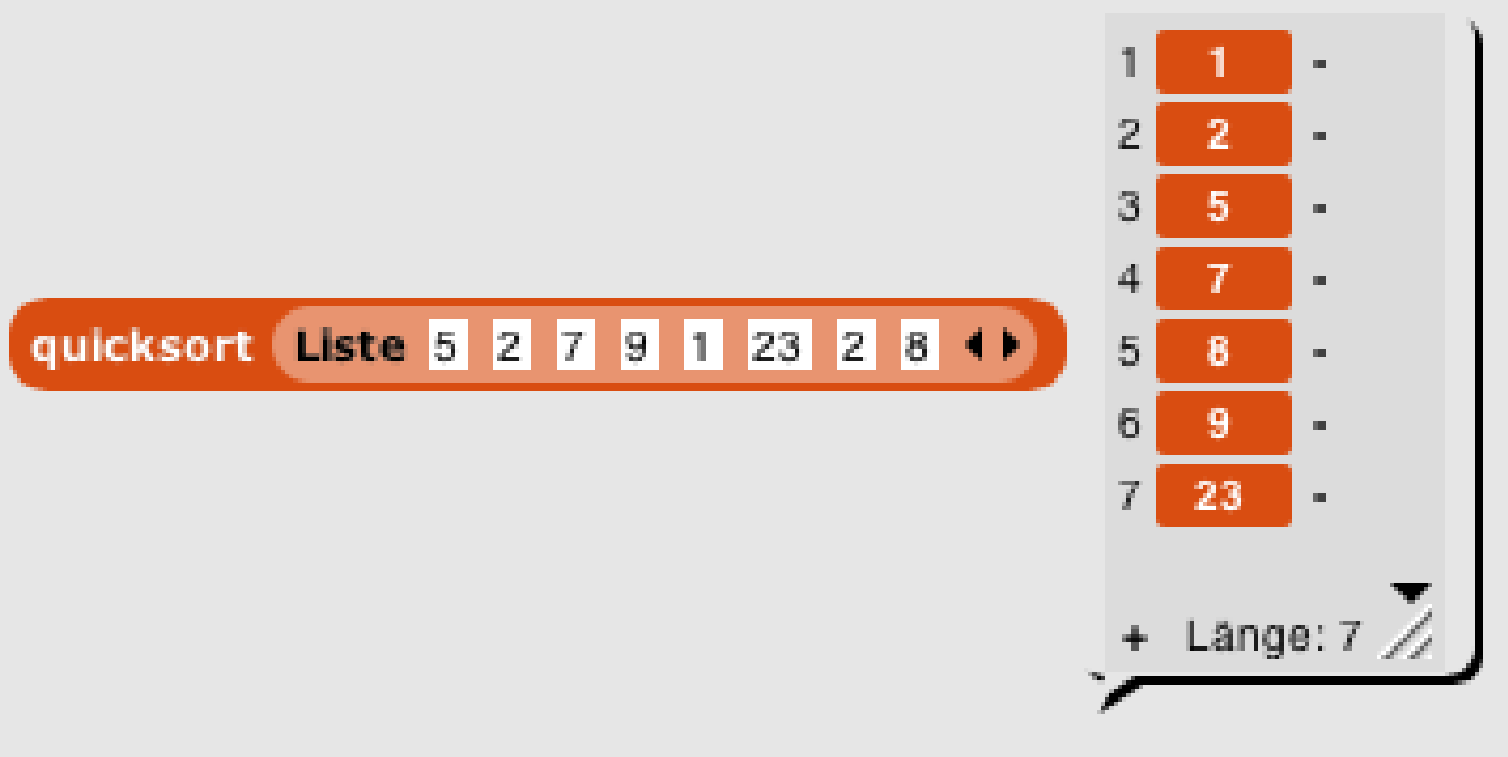




- Umringte Blöcke mit Leerstellen (hier z.B.: □+10) als anonyme Funktionen bzw. ***Lambda-Ausdrücke***
- Klassische Funktionen zur Listenverarbeitung in *Tools*-Bibliothek:



Quicksort für Eilige



Quicksort – wie war das noch?

1. Eine leere Liste ist natürlich bereits sortiert. Ansonsten:
2. Wähle ein beliebiges Element aus der Liste, das sogenannte Pivot-Element *p*
3. Bilde je eine Liste mit den Elementen,
   - die kleiner sind als das Pivot-Element: *l*
   - die größer sind als das Pivot-Element: *h*
4. Sortiere beide Listen nach dem selben Prinzip
5. Setze *l, p* und *h* zu einer Liste zusammen – diese ist nun sortiert!

Wie viel Code benötigen wir wohl, um das umzusetzen?
In Java? In *Snap*?

Quicksort für Eilige

Viel kürzer geht es nicht mehr!

☺

Links zu den Programmbeispielen

- Funktionen als Parameter von Funktionen: http://is.gd/higherOrderFunctions0
- Funktionen als Listenelemente: http://is.gd/higherOrderFunctions1
- *map*, *filter*, *reduce* mit Lambda-Ausdrücken: http://is.gd/higherOrderFunctions2
- Quicksort: http://is.gd/Quicksort

Jenseits von Java & Co.

Snap! kennt diese Kontrollstruktur:

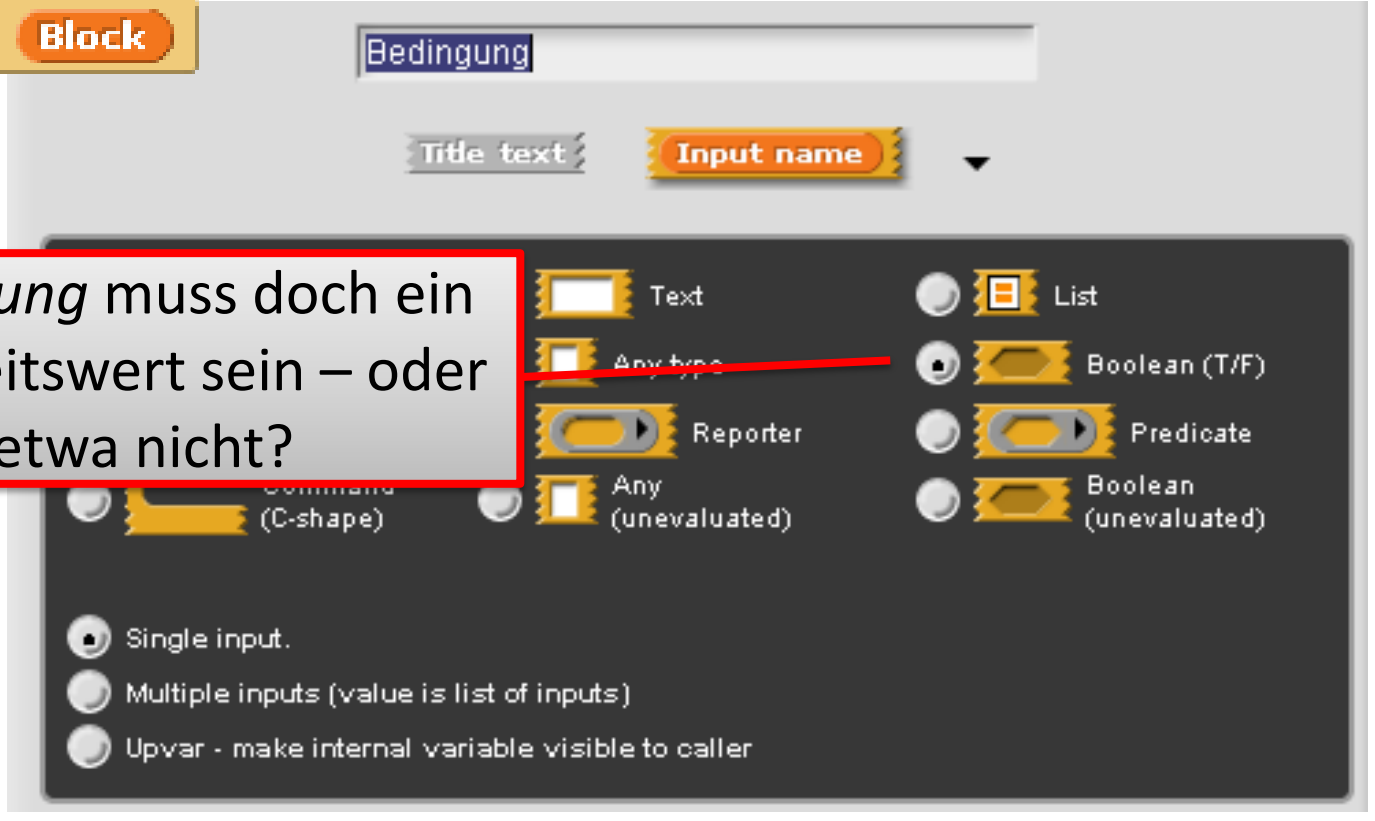
Ja, ist das denn die Möglichkeit:
Snap! hat keinen Block „wiederhole solange“ bzw. ***while*** !!!

Dann bauen wir uns eben selbst einen ***While***-Block!!!

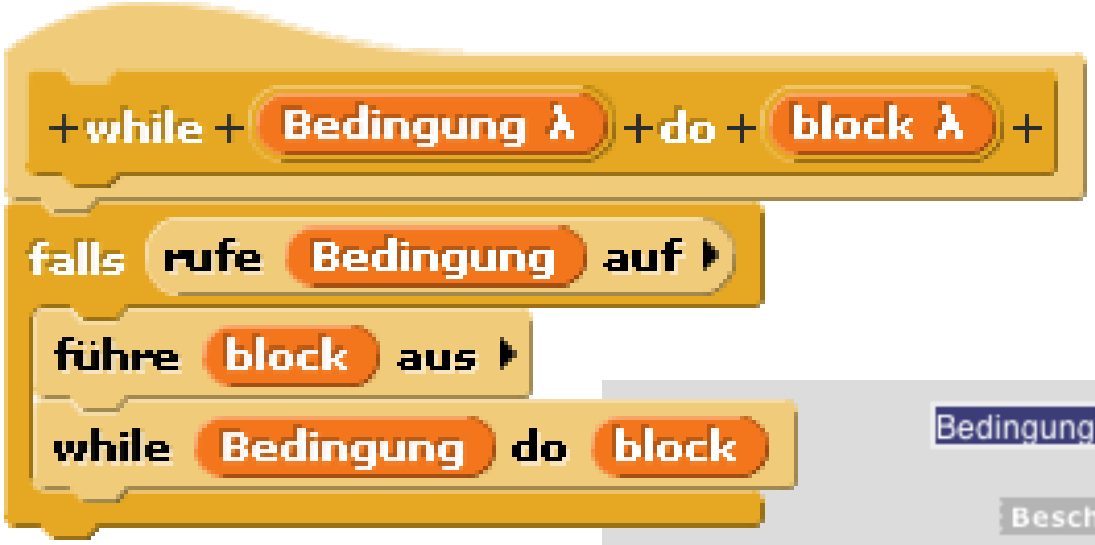
Die While-Schleife als Eigenbau

Fast richtig – aber so funktioniert die Schleife noch nicht. Warum?

Bedingung muss doch ein Wahrheitswert sein – oder etwa nicht?

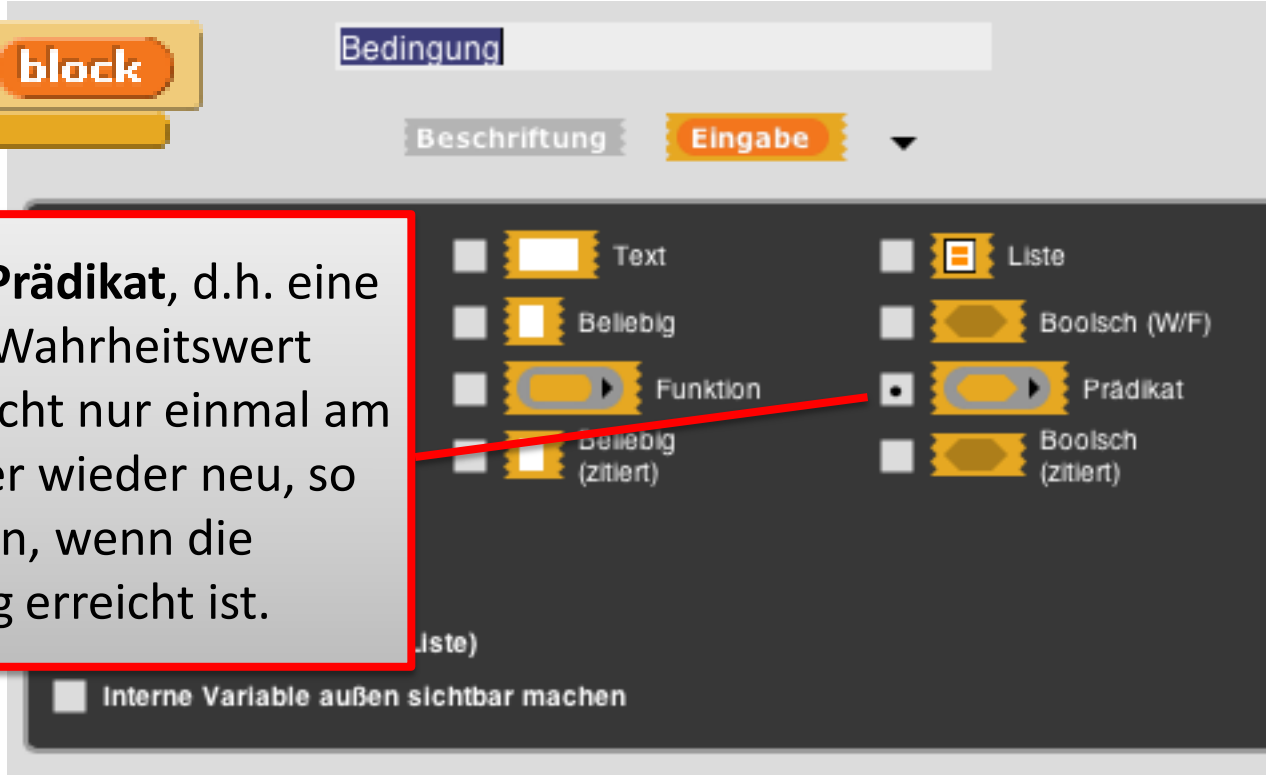


Die While-Schleife als Eigenbau



So stimmt‘s! Sehen Sie den Unterschied?

Bedingung ist jetzt ein **Prädikat**, d.h. eine Funktion, die einen Wahrheitswert *berechnet* – und zwar nicht nur einmal am Anfang, sondern immer wieder neu, so dass wir feststellen, wenn die Abbruchbedingung erreicht ist.



Eine programmierbare Programmiersprache

- Auch andere Konstrukte, die in anderen Programmiersprachen zur fest vorgebenen Syntax gehören – oder eben nicht! – kann man sich in *Snap!* einfach **selbst bauen**, z.B.:
  - eine *for*-Schleife
  - das *break*-Statement zum Verlassen von Schleifen
  - Threads
  - Objektorientierung
- Es stimmt also: *„Snap! is Scheme disguised as Scratch.“*

Links zu den Programmbeispielen

- While-Schleife mit Fehler: http://is.gd/WhileSchleifeInkorrekt
- Funktionierende While-Schleife: http://is.gd/WhileSchleife
- For-Schleife: http://is.gd/ForSchleife

Zusammenfassung

- *Snap!* ist
  - intuitiv wie Scratch → ideal für Anfänger
  - mächtig wie Scheme → ideal für Fortgeschritten(st)e
- Komplexe Konzepte in kurzer Zeit vermittelbar
- Nicht explizit objektorientiert – aber OO möglich

  → Hinterfragen: Warum genau unterrichten wir OO?
- *Snap!*
  - muss nicht installiert werden
  - läuft demnächst garantiert auch auf Ihrem Tablet
- *Snap!* macht Spaß!

Weiterführendes

- *Snap!* Hauptseite: http://snap.berkeley.edu
- Handbuch: http://snap.berkeley.edu/SnapManual.pdf
- Snap! ist freie, offene Software; veröffentlicht unter der AGPL. Quellcode: https://github.com/jmoenig/Snap--Build-Your-Own-Blocks
- Falls Sie Fehler entdecken: https://github.com/jmoenig/Snap--Build-Your-Own-Blocks/issues

Vielen Dank!

Über Fragen und Feedback freue ich mich:

Michael Brenner
micbre@gmail.com
http://zurueckindieschule.wordpress.com