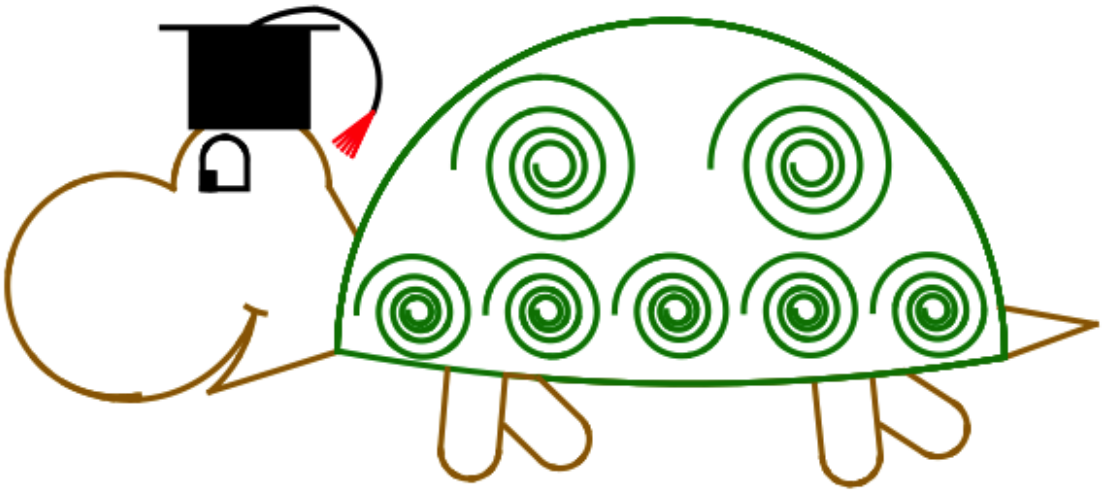
ABZ AUSBILDUNGS- UND BERATUNGSZENTRUM
FÜR INFORMATIKUNTERRICHT

Hans-Joachim Böckenhauer Juraj Hromkovič Dennis Komm

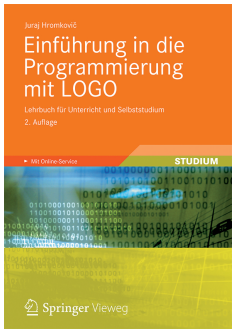
Programmieren mit LOGO
für Fortgeschrittene



ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Programmieren mit LOGO

Dieses Skript ist eine gekürzte Version der Lektionen 9 bis 11 des Lehrbuches *Einführung in die Programmierung mit LOGO*. Das Lehrbuch enthält viele weitere Aufgaben und Erklärungen. Ausserdem ist es mit Hinweisen für die Lehrperson versehen. Das Lehrbuch umfasst insgesamt 15 Lektionen.



Juraj Hromkovič. *Einführung in die Programmierung mit LOGO: Lehrbuch für Unterricht und Selbststudium*. 2. Aufl., Springer Vieweg 2012. ISBN: 978-3-8348-1852-2.

Version 0.3, 0. 0, SVN-Rev: -2

Programmierumgebung

Die vorliegenden Unterrichtsunterlagen wurden für die Programmierumgebung XLogo entwickelt. XLogo ist auf der Webseite `xlogo.tuxfamily.org` kostenlos verfügbar.

Damit die Logo-Programme aus den Unterlagen ausgeführt werden können, muss XLogo auf Englisch eingestellt werden.

Nutzungsrechte

ABZ

Das Ausbildungs- und Beratungszentrum für Informatikunterricht der ETH Zürich unterstützt Schulen und Lehrkräfte, die ihren Informatikunterricht entsprechend auf- oder ausbauen möchten, mit einem vielfältigen Angebot. Es reicht von individueller Beratung und Unterricht durch ETH-Professoren und das ABZ-Team direkt vor Ort in den Schulen über Ausbildungs- und Weiterbildungskurse für Lehrkräfte bis zu Unterrichtsmaterialien.

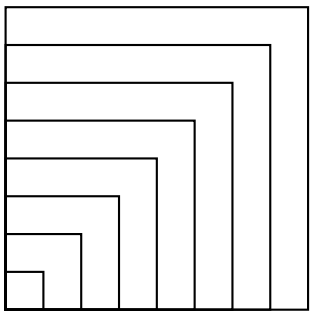
`www.abz.inf.ethz.ch`

8 Programme mit Variablen

In Kapitel 5 des ersten Teils des Skriptes haben wir gesehen, wie man Programme schreibt, die einen Parameter benutzen. Wir haben hier beispielsweise gelernt, wie wir ein Quadrat zeichnen können, dessen Seitenlänge durch einen Parameter `:GR` bestimmt werden kann.

```
to QUADRAT :GR
repeat 4 [fd :GR rt 90]
end
```

Nehmen wir nun an, wir wollten 8 Quadrate mit jeweils verschiedenen Grössen zeichen, die eine gemeinsame Ecke unten links besitzen. Dabei soll das kleinste Quadrat eine Seitenlänge von 10 besitzen, das zweite eine von 20, das dritte eine von 30 usw.



Ein solches Bild können wir beispielsweise mit dem folgenden Programm `ACHTQUADRATE` erstellen, das einfach unser Programm `QUADRAT` 8-mal als Unterprogramm aufruft und dabei jeweils einen anderen Wert für `:GR` übergibt.

```
to ACHTQUADRATE
QUADRAT 10
QUADRAT 20
QUADRAT 30
QUADRAT 40
QUADRAT 50
QUADRAT 60
QUADRAT 70
QUADRAT 80
end
```

Diese Lösung stellt uns allerdings nicht zufrieden, da hier ein grosser Teil des Programms aus Wiederholungen besteht. Insbesondere würde ein ähnliches Programm `TAUSENDQUADRATE`, das 1000 Quadrate auf diesselbe Art zeichnet, aus 1000 Zeilen bestehen, die sich kaum unterscheiden (probiere es *nicht* aus).

Um die Programmlänge zu reduzieren und Tipparbeit zu sparen, würde sich wieder anbieten, den **`repeat`**-Befehl, den wir in Kapitel 2 kennengelernt haben, zu benutzen. Wie aber sieht der zu wiederholende Teil aus?

Die im Folgenden vorgestellte Idee besteht darin, dem Unterprogramm `QUADRAT` einen Parameter `:VARGR` zu übergeben, dessen Wert sich im Verlauf des Programms ändert. Diesen Parameter wollen wir innerhalb des Programms `ACHTQUADRATE` definieren, er wird `ACHTQUADRATE` also nicht als Parameter übergeben.

Definiere einen Parameter `:VARGR` und gib ihm zunächst den Wert 10.
`QUADRAT :VARGR`
Erhöhe `:VARGR` um 10
} 8-mal

Die zentrale Frage ist also, wie wir das Definieren und Erhöhen von `:VARGR` realisieren können. Bislang haben wir Parametern ausdrücklich einen Wert zugewiesen. Jetzt wollen wir diesen Wert während der Ausführung gewissermassen vom Computer ändern lassen. Wir sprechen deswegen in diesem Zusammenhang von einer **Variablen**.

Zu diesem Zweck lernen wir den Befehl **`make`** kennen, mit dem wir genau dies erreichen können. Um beispielsweise den Wert einer Variablen `:X` auf 10 zu setzen, geben wir

```
make "X 10
```

ein. Hierbei ist zu beachten, dass wir, wenn wir einer Variablen einen Wert zuweisen, `"X` schreiben und nicht `:X`. Wichtig ist ausserdem, dass die Variable `:X` nicht bereits existieren muss. Gibt es eine solche Variable noch nicht, so wird sie mit **`make`** automatisch erzeugt und der Wert (hier 10) wird ihr zugewiesen.

Wir können für das Zuweisen nicht nur konkrete Zahlen, sondern auch andere Variablen benutzen. Somit setzt

```
make "X :Y
```

den Wert von `:X` auf den Wert von `:Y`. Es ist sogar möglich, komplexere **arithmetische Ausdrücke** zu benutzen, also Rechnungen. Der Befehl

```
make "X :Y*5+9
```

weist `:X` den Wert von `:Y` mit 5 multipliziert und zu 9 addiert zu. Dies ist für uns von besonderer Bedeutung, denn wir können hierdurch den Wert einer Variable vergrössern oder verringern. Wenn `:X` zum Beispiel anfangs den Wert 7 besitzt, so ist dieser nach dem Ausführen von

```
make "X :X+9
```

auf 16 gesetzt. Der alte Wert 7 wurde also überschrieben. Wir können natürlich auch hier kompliziertere Ausdrücke wie beispielsweise

```
make "X (2+9.6)*(1.2*:X+9)*7
```

verwenden. Der Computer kümmert sich darum, dass die rechte Seite zunächst korrekt berechnet wird, und weist das Ergebnis danach `:X` zu.

Wir halten an dieser Stelle noch einmal fest, dass die Werte von Variablen sich im Laufe des Programms ändern können. Sie müssen hierzu nicht in der **to**-Zeile des Programms definiert werden, sondern können im Programm mit **make** „erzeugt“ werden. Für Variablen, die ihren Wert nicht ändern, verwenden wir weiterhin auch die Bezeichnung Parameter.

Jetzt können wir unser Programm `ACHTQUADRATE` schreiben, indem wir dem Unterprogramm `QUADRAT` die Variable `:VARGR` übergeben, deren Wert wir nach jeder Wiederholung vergrössern. Zu Beginn definieren wir `:VARGR` und setzen ihren Wert auf 10.

```
to ACHTQUADRATE
make "VARGR 10
repeat 8 [QUADRAT :VARGR make "VARGR :VARGR+10]
end
```

Führe dieses Programm zunächst aus und überprüfe, ob alles richtig dargestellt wird.

Aufgabe 1

Schreibe ein Programm `ELFQUADRATE`, das ähnlich wie `ACHTQUADRATE` funktioniert, aber 11 Quadrate zeichnet, deren Seitenlänge jeweils um 8 wächst, wobei das kleinste Quadrat eine Seitenlänge von 15 besitzt.

Aufgabe 2

Schreibe ein Programm `QUADRATE` mit einem Parameter `:ANZ`, das `:ANZ` Quadrate zeichnet, wobei das kleinste Quadrat eine Seitenlänge von 20 besitzt und sich die Seitenlängen aufeinander folgender Quadrate um 5 unterscheiden.

Aufgabe 3

Erweitere `QUADRATE` zu einem Programm `QUADRATE2`, bei dem du zusätzlich die Seitenlänge des kleinsten Quadrates mit einem Parameter `:START` angeben kannst.

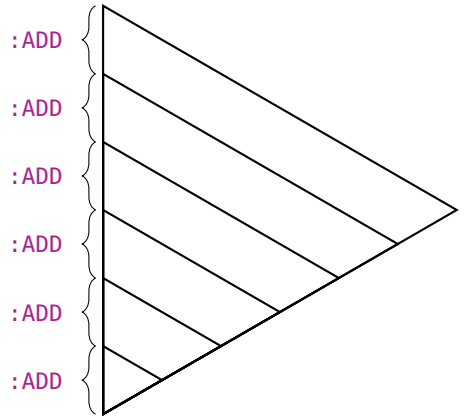
In Kapitel 4 haben wir bereits gelernt, wie wir regelmässige Vielecke zeichnen können. Hierbei werden 360° auf die einzelnen Drehungen verteilt, sodass die Schildkröte am Ende wieder auf ihrer Ausgangsposition steht. Das folgende Programm zeichnet ein Dreieck.

```
to DREIECK :GR
repeat 3 [fd :GR rt 120]
end
```

Jetzt wollen wir Bilder aus verschiedenen Dreiecken zusammensetzen.

Aufgabe 4

Schreibe ein Programm SECHSDREIECKE, das das folgende Bild zeichnet, das aus 6 Dreiecken besteht. Die Seitenlänge soll dabei jeweils um :ADD wachsen, wobei :ADD ein frei wählbarer Parameter ist, der dem Programm übergeben wird. Das kleinste Dreieck soll ausserdem eine Seitenlänge von :ADD besitzen.



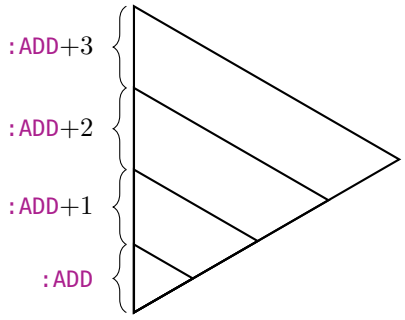
Analog zu den Programmen, die mehrere Quadrate zeichnen, soll SECHSDREIECKE DREIECK als Unterprogramm verwenden.

Aufgabe 5

Erweitere SECHSDREIECKE zu einem Programm DREIECKE, bei dem die Anzahl der Dreiecke mit einem weiteren Parameter :ANZ angegeben werden kann. Ausserdem sollen die Dreicke nicht mehr konstant um :ADD wachsen, sondern wie folgt. Das erste Dreieck besitzt eine Seitenlänge von :ADD, die des zweiten Dreiecks ist um :ADD+1 grösser, die des dritten um :ADD+2 usw.

Wird DREIECKE also mit dem Wert 4 für :ANZ aufgerufen, soll dies zu folgender

Darstellung führen.



Aufgabe 6

Mit dem folgenden Programm NEUNQUADRATE sollen 9 Quadrate gezeichnet werden, wobei das kleinste eine Seitenlänge von 10 besitzt und die Seitenlänge dann immer um 10 erhöht wird. Es befinden sich zwei Fehler darin. Kannst du sie finden und beheben?

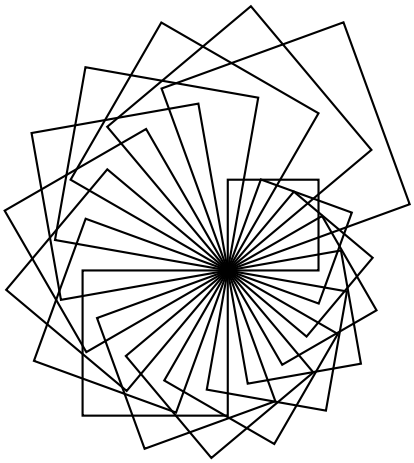
```
to NEUNQUADRATE
repeat 9 [QUADRAT :VARGR make :VARGR :VARGR+10]
end
```

Schreibe die korrigierte Version in die folgenden Felder.

Aufgabe 7

Das folgende Bild besteht aus Quadraten, die grösser werden, wobei der Winkel zwischen zwei aufeinanderfolgenden Quadraten immer gleich gross ist, sodass sie

gleichmässig auf einem Kreis verteilt sind.



Erstelle ein Programm `ROTQUAD`, das derartige Bilder zeichnet. Hierbei sollen die Grösse des kleinsten Quadrates, die additive Änderung, um die sich die Quadrate vergrössern, und die Anzahl der Quadrate frei wählbar sein. Die Quadrate sollen gleichmässig auf die ganze Drehung der Schildkröte verteilt werden.

Hinweis: Berechne zuerst aus der Anzahl der Quadrate, um wie viel Grad du die Schildkröte in jedem Schritt drehen musst, indem du 360 durch diese Anzahl teilst. Verwende das Programm `QUADRAT` als Unterprogramm.

Additive und multiplikative Änderungen

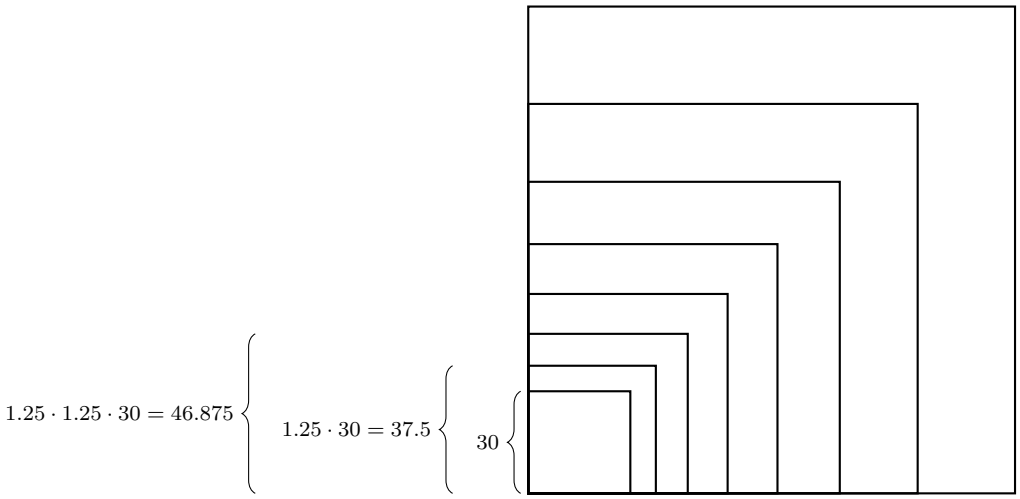
Bislang haben wir die Seitenlängen der Quadrate und Dreicke um einen **additiven Term** (eine additive Änderung) wachsen lassen, das heisst, die Werte von Variablen wurden jeweils durch *Additionen* vergrössert. Nun möchten wir uns etwas komplizierteren Bildern widmen, bei denen multiplikative Änderungen (sogenannte **Faktoren**) benutzt werden.

Die Idee, die für die Umsetzung benötigt wird, haben wir schon kennengelernt, denn wir wissen, dass mit dem **`make`**-Befehl beliebige arithmetische Ausdrücke benutzt werden dürfen. Folglich können wir mit den Befehlen

```
make "VARGR 30
repeat 8 [QUADRAT :VARGR make "VARGR :VARGR*1.25]
```

das folgende Bild zeichnen, wobei für jedes Quadrat die Seitenlänge um einen Faktor von

1.25 grösser ist als die des vorangehenden.



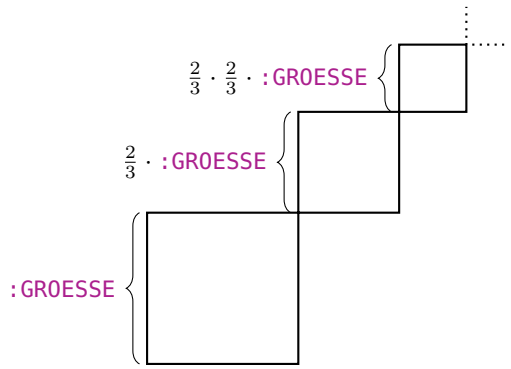
Erstelle als nächstes ein Programm `ACHTQUADRATEMULT`, das aus dem oben beschriebenen Code besteht und teste es.

Aufgabe 8

Erweitere dein Programm `ACHTQUADRATEMULT` zu einem Programm `QUADRATEMULT`, sodass die multiplikative Änderung (1.25) und die Anzahl der Quadrate (8) durch Parameter gewählt werden können.

Aufgabe 9

Bis jetzt hatten alle gezeichneten Quadrate einen gemeinsamen Ursprung in der unteren linken Ecke. Nun wollen wir eine Treppe von Quadraten zeichnen. Das erste Quadrat hat hierbei eine frei wählbare Seitenlänge von `:GROESSE` und das jeweils folgende Quadrat hat eine Seitenlänge, die genau zwei Dritteln der Seitenlänge des vorangehenden Quadrates entspricht.



Schreibe ein Programm `QUADTREPPE`, das zwei Parameter `:GROESSE` und `:STUFEN` erhält, wobei `:GROESSE` die Seitenlänge der ersten Stufe angibt und `:STUFEN` die Gesamtanzahl der Stufen.

Aufgabe 10

Erweitere `QUADTREPPE` nun wiederum zu einem Programm `QUADTREPPEMULT`, bei dem der multiplikative Faktor mit einem Parameter `:MULT` frei gewählt werden kann.

Wir stellen fest, dass die Bilder, wenn wir multiplikative Faktoren benutzen, viel schneller sehr gross beziehungsweise klein werden. Woran liegt das? Betrachten wir wieder additive Änderungen. Nehmen wir an, dass wir ein Quadrat zeichnen wollen, das in jedem Schritt der **`repeat`**-Schleife um 20 Punkte wächst, wobei das erste Quadrat eine Seitenlänge von 10 besitzt. Die Quadrate haben also die Seitenlängen

$$10, 30, 50, 70, 90, 110, 130, 150, 170, \ldots.$$

Vom ersten zum zweiten Quadrat vergrössert sich die Seitenlänge also um einen multiplikativen Faktor von

$$\frac{30}{10} = 3.$$

Vom zweiten zum dritten Quadrat berechnen wir einen multiplikativen Faktor von

$$\frac{50}{30} = 1.\overline{6}$$

und vom dritten zum vierten

$$\frac{70}{50} = 1.4$$

und diese Verhältnisse werden immer kleiner. Wir haben es hier also mit multiplikativen Faktoren zu tun, die sich mit fortlaufenden Wiederholungen der **repeat**-Schleife verkleinern und zwar so, dass sie sich der 1 annähern.

Wenn wir jedoch ausdrücklich einen multiplikativen Faktor benutzen, so ist dieser in jedem Zeitschritt gleich, weswegen sich die Zahlen viel schneller vergössern. Starten wir beispielsweise wieder mit einer Quadratgrösse von 10 und benutzen einen multiplikativen Faktor von 2, so ergeben sich die Seitenlängen

$$10, 20, 40, 80, 160, 320, 640, 1280, 2560, \ldots$$

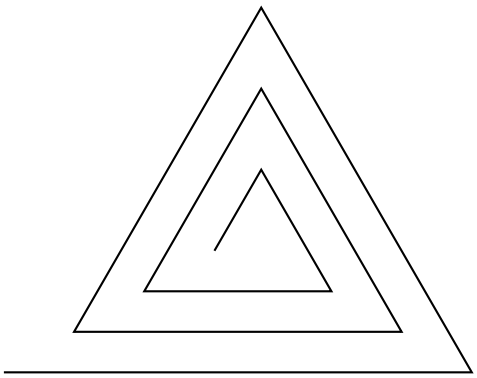
und bereits nach 300 Wiederholungen entspricht dies einer Zahl mit über 90 Dezimalstellen. Dies ist eine grössere Zahl als die vermutete Anzahl an Elementarteilchen in unserem Universum.

Auf der anderen Seite sind wir bei der ersten Reihe von Seitenlängen mit einer additiven Änderung nach 300 Wiederholungen lediglich bei einer Seitenlänge von 6010.

Das Wachstum mit einem konstanten multiplikativen Faktor wird von Wissenschaftlern als **exponentielles Wachstum** bezeichnet und wir treffen darauf sehr häufig bei Phänomenen, die in der Natur auftreten, zum Beispiel bei der Vermehrung von Bakterien.

Aufgabe 11

Schreibe ein Programm `SCHNECKE`, das das folgende Bild von innen nach aussen zeichnet. Hierbei sollen die Länge der ersten kürzesten Linie und die Differenz, also der additive Unterschied, zwischen zwei aufeinanderfolgenden Linien über zwei Parameter angegeben werden.



Aufgabe 12

Schreibe ein Programm SCHNECKEMULT, das eine Schnecke wie bei SCHNECKE zeichnet, aber einen multiplikativen Faktor verwendet, um die Linien länger werden zu lassen.

Verschiedene Grössen ineinander umrechnen

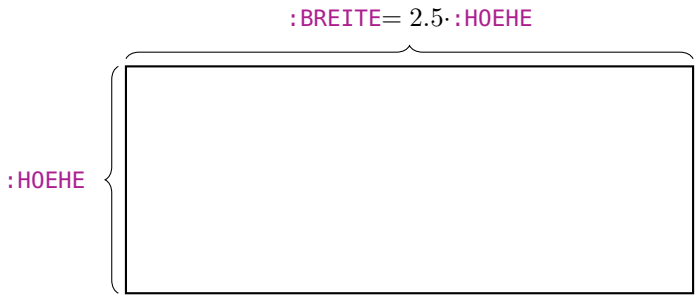
Bislang haben wir, um die Grösse eines Objektes zu bestimmen, die Seitenlängen angegeben. Jetzt wollen wir den **make**-Befehl benutzen, um beispielsweise durch die Angabe des Umfangs die Seitenlängen auszurechnen, sodass wir ein Objekt darstellen können.

Betrachten wir ein Program QUADUM, das ein Quadrat zeichnen soll, welches einen Umfang besitzt, der mit einem Parameter :UM angegeben werden kann.

Da der Umfang sich auf vier Seiten verteilt, die alle dieselbe Länge haben, können wir das Program wie folgt schreiben.

```
to QUADUM :UM
make "SEITE :UM/4
repeat 4 [fd :SEITE rt 90]
end
```

Wie sieht es aber aus, wenn wir keine Quadrate betrachten, sondern allgemeine Rechtecke? In diesem Fall benötigen wir mehr Informationen, denn wir müssen wissen, welches Verhältnis die langen und kurzen Seitenlängen zueinander besitzen. Betrachten wir zum Beispiel das folgende Rechteck, bei dem die horizontalen Seiten 2.5-mal länger sind als die vertikalen.



Wir sehen, dass sich für den Parameter :UM, der den Umfang angibt,

$$:\texttt{UM} = 2 \cdot :\texttt{HOEHE} + 2 \cdot :\texttt{BREITE} = 2 \cdot :\texttt{HOEHE} + 2 \cdot 2.5 \cdot :\texttt{HOEHE} = 7 \cdot :\texttt{HOEHE}$$

ergibt und somit

$$:\texttt{HOEHE} = \frac{:\texttt{UM}}{7}.$$

Aufgabe 13

Benutze diese Beobachtungen, um ein Programm RECHTUM zu entwickeln, das einen Parameter :UM erhält und ein Rechteck zeichnet, das ein Seitenverhältnis von 2.5 zu 1 und einen Umfang von :UM besitzt. Dabei sollen die horizontalen Seiten die längeren sein.

Aufgabe 14

Jetzt möchten wir ein Programm erstellen, das nicht den Umfang, sondern die Fläche gegeben bekommt. Schreibe ein Programm QUADFL, das einen Parameter :FLAECHE erhält und ein Quadrat mit der angegebenen Fläche zeichnet.

Hinweis: Benutze den Befehl **sqrt**, der die Quadratwurzel eines gegebenen arithmetischen Ausdrucks berechnet. Zum Beispiel bewegt also **fd sqrt** 100 die Schildkröte um 10 Schritte nach vorne und **make** "X **(sqrt** 625)-1 setzt den Wert der Variablen :X auf 24. Die Klammerung ist hierbei wichtig.

Aufgabe 15

Erweitere das Programm QUADFL zu einem Programm RECHTECKFL, das zwei Parameter :FLAECHE und :VERH erhält. Hierbei gibt :VERH das Seitenverhältnis und :FLAECHE die Fläche des zu zeichnenden Rechtecks an. Diesmal sollen die vertikalen Seitenlängen grösser sein und wir gehen immer davon aus, dass :VERH eine Zahl grösser 1 ist.

Hinweis: Benutze hierbei, dass für Variablen, die die beiden Seitenlängen repräsentieren, :FLAECHE=:HOEHE*:BREITE=:HOEHE*:VERH*:HOEHE ist.

Wir haben in Kapitel 4 gelernt, einen Kreis zu zeichnen, indem wir das folgende Programm benutzen.

```
to KREIS :SCHRITTLN
repeat 360 [fd :SCHRITTLN rt 1]
end
```

Dabei macht die Schildkröte in jeder der 360 Wiederholungen genau :SCHRITTLN viele Schritte, also insgesamt 360·:SCHRITTLN, was genau dem Umfang des Kreises entspricht.

Möchten wir also nun ein Programm KREISUM schreiben, das den Umfang des Kreises als Parameter übergeben bekommt, so gehen wir einfach wie folgt vor.

```
to KREISUM :UM
repeat 360 [fd :UM/360 rt 1]
end
```

In Kapitel 4 haben wir ausserdem gelernt, wie wir mit dem Befehl **setpc** die Farbe des Stiftes ändern können. Hierbei werden Farben durch Zahlen repräsentiert, die hier noch einmal zusammengefasst sind.

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

Der **make**-Befehl erlaubt es uns nun, Farbnummern in einer Variablen zu speichern, sodass wir zum Beispiel eine bunte Linie mit dem Programm

```
to BUNTELINIE
make "FARBE 0
rt 90
repeat 8 [setpc :FARBE fd 30 make "FARBE :FARBE+1]
end
```

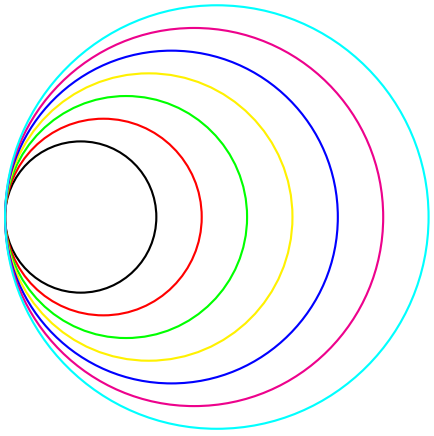
zeichnen können.

Aufgabe 16

Schreibe ein Programm `AUGE`, das drei Parameter `:ANZ`, `:START`, `:ABSTAND` erhält. Dabei soll `:START` den Umfang des kleinsten Kreises bestimmen und die folgenden Kreise sollen einen Umfang besitzen, der jeweils um eine additive Änderung von `:ABSTAND` wächst. Insgesamt sollen `:ANZ` Kreise gezeichnet werden, wobei `KREISUM` als Unterprogramm verwendet wird.

Ferner sollen die Kreise alle verschiedene Farben haben. Du darfst hierbei davon ausgehen, dass der Wert des Parameters `:ANZ` nie grösser als 16 ist.

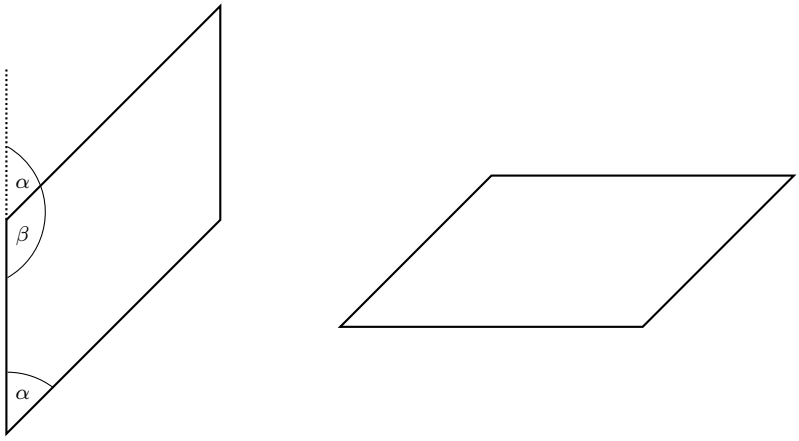
Durch den Aufruf `AUGE 7 400 80` soll also das folgende Bild erzeugt werden.



Optische Täuschungen

Wir haben gesehen, dass wir bei den Programmen, die Rechtecke zeichnen, in der Regel mehr rechnen müssen als bei Programmen, die Quadrate zeichnen. Das liegt daran, dass Rechtecke Verallgemeinerungen von Quadraten sind. Quadrate sind nämlich genau die Rechtecke, bei denen das Seitenverhältnis 1 zu 1 ist.

Eine weitere Verallgemeinerung stellen Parallelogramme dar (ein solches haben wir schon in Kapitel 5 gezeichnet), bei denen, anders als bei *Recht*ecken, nicht alle Winkel *rechte* Winkel sein müssen, sondern lediglich gegenüberliegende Winkel gleich, weswegen jeweils gegenüberliegende Seiten parallel sind.



Um ein Parallelogramm zeichnen zu können, brauchen wir eine weitere Information, nämlich die Grösse eines Winkels α von zwei verbundenen Seiten. Alle anderen Winkel können dann aus diesem berechnet werden: Der Winkel an der gegenüberliegenden Ecke hat immer dieselbe Grösse und die beiden anderen Winkel ergeben sich ganz einfach zu

$$180 - \alpha.$$

Weiterhin sollten wir uns darauf einigen, wie wir ein Parallelogramm auf dem Bildschirm zeichnen. Wir wollen dies so tun, dass die linke und die rechte Seite parallel zu der linken und rechten Monitorkante sind, also so wie das linke oben dargestellte Bild. Tatsächlich zeigt das rechte Bild dasselbe Parallelogramm, aber diesmal so, dass die obere und die untere Seite parallel zu der oberen und unteren Monitorkante sind.

Wie wird ein solches Parallelogramm nun gezeichnet? Zunächst geht die Schildkröte von der Startposition eine gewisse Anzahl Schritte, die genau der linken Seite entspricht, nach oben. Nun soll der Innenwinkel der ersten zu zeichnenden Ecke β Grad betragen, weswegen sie sich um $180 - \beta = \alpha$ Grad nach rechts drehen muss usw.

Aufgabe 17

Jetzt wollen wir ein Programm erstellen, das ein Parallelogramm bei gegebenem Winkel α und Umfang zeichnet. Das Seitenverhältnis soll hierbei 2 zu 1 sein, wobei die vertikalen Linien die längeren sind.

Vervollständige hierzu den folgenden Programmcode an den durch ... markierten Stellen und gib ihn in deinen Editor ein.

```
to PARALLELOGRAMM :ALPHA :UM
make "KURZESEITE ...
make "LANGESEITE ...
make "BETA ...
repeat 2 [fd :LANGESEITE rt :ALPHA fd :KURZESEITE rt :BETA]
end
```

Teste dein Programm mit verschiedenen Werten für die beiden Parameter :ALPHA und :UM, bis du sicher bist, dass es das erwünschte Bild darstellt.

In Kapitel 7 haben wir gelernt, wie wir mit dem Befehl **wait** Animationen erstellen können. Jetzt benutzen wir PARALLELOGRAMM, um eine sich öffnende Tür zu animieren.

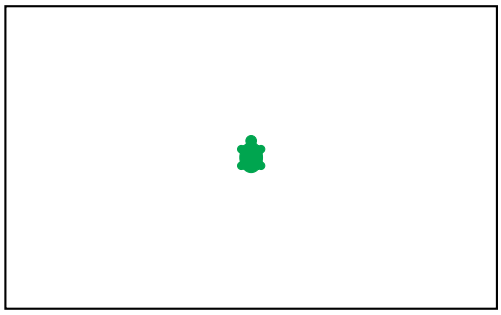
Gib das folgende Programm in deinen Editor ein und teste es.

```
to TUER
ht
make "WINKEL 90
repeat 45 [
  setpw 1 ppt PARALLELOGRAMM :WINKEL 300
  wait 4
  setpw 10 pe PARALLELOGRAMM :WINKEL 300
  make "WINKEL :WINKEL+2
]
end
```

Zum Radieren benutzen wir hier einen grösseren Stiftdurchmesser, damit bei der Animation keine Spur entsteht. Wir können diesen Durchmesser mit dem Befehl **setpw** angeben.

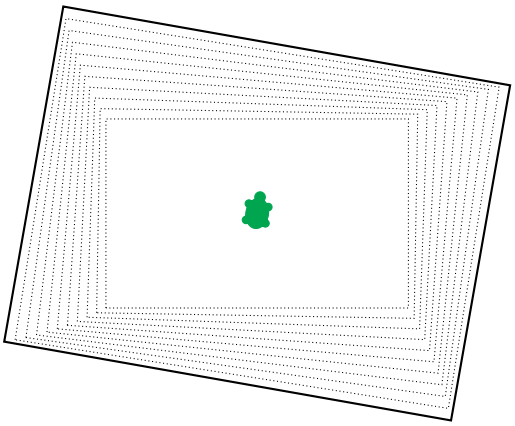
Aufgabe 18

Schreibe ein Programm `RECHTMITTE`, das ein Rechteck zeichnet, für das Umfang und Seitenverhältnis gegeben sind. Die horizontalen Seiten sollen hierbei die längeren sein. Wichtig ist, dass die Schildkröte vor und nach dem Zeichnen des Rechtecks in dessen Mitte steht und nach oben guckt.



Aufgabe 19

Benutze jetzt `RECHTMITTE` als Unterprogramm in einem Programm `RECHTDREHUNG`, das ein Rechteck so animiert, dass es sich auf den Betrachter zu bewegt, also grösser wird, und sich hierbei um die eigene Achse dreht.



Dabei sollen der Umfang des kleinsten Rechtecks, das Seitenverhältnis der Rechteckseiten, die additive Änderung, um die die Rechtecke grösser werden, der Winkel, um den gedreht wird, die Anzahl der Drehungen und die Wartezeit zwischen den Drehungen durch Parameter in dieser Reihenfolge angegeben werden können.

Teste dein Programm mit `RECHTDREHUNG 100 1.5 7 1 500 1`.

Hinweis: Benutze den Befehl **`ht`** zu Beginn des Programms, damit die Schildkröte unsichtbar ist.

9 Register des Rechners

In diesem Kapitel wollen wir uns mit den Prozessen beschäftigen, die innerhalb des Rechners ablaufen, wenn dieser Parameter und Variablen verwaltet. Die gewonnenen Einsichten werden uns später dabei helfen, Fehler zu vermeiden, wenn wir komplexere Programme erstellen, die Unterprogramme verwenden.

Parameter

Betrachten wir wieder das Programm QUADRAT, dem wir einen Parameter :GR übergeben, dessen Wert anschliessend die Seitenlänge des gezeichneten Quadrates bestimmt.

```
to QUADRAT :GR
repeat 4 [fd :GR rt 90]
end
```

Das Programm **repeat** 4 [**fd** :GR **rt** 90] wird also unter dem Namen QUADRAT gespeichert. Zusätzlich merkt der Rechner sich, dass dieses Programm einen Parameter :GR besitzt und reserviert hierfür einen Speicherplatz, den wir **Register** nennen. Diesem Register wird der Name GR(QUADRAT) gegeben, der die folgende Bedeutung besitzt: „In diesem Register ist der Wert des Parameters :GR aus dem Programm QUADRAT gespeichert."

Wird nun der Befehl

```
QUADRAT 35
```

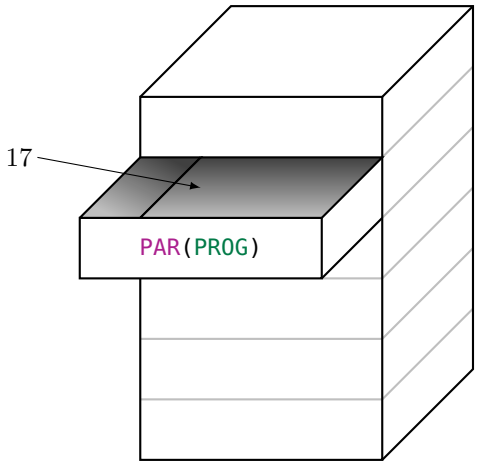
ausgeführt, so ruft der Rechner das entsprechende Programm auf. Anschliessend speichert er den Wert 35 im reservierten Register GR(QUADRAT). Bei der Ausführung des Programms wird für jedes Vorkommen von :GR der im Register GR(QUADRAT) gespeicherte Wert eingesetzt, sodass tatsächlich **repeat** 4 [**fd** 35 **rt** 90] ausgeführt und somit ein Quadrat der Grösse 35×35 gezeichnet wird.

Für den Rechner bedeutet dies, viermal die Abfolge der Befehle

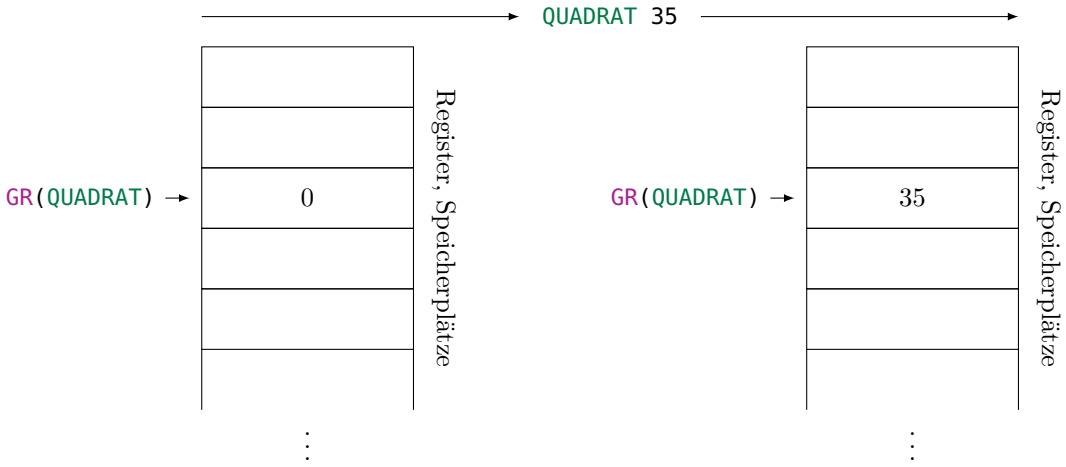
fd „Der aktuelle Wert im Register GR(QUADRAT)" **rt** 90

auszuführen. Wichtig ist, dass ein Register, dem wir einen Namen gegeben haben, nie leer ist, sondern eine 0 enthält, wenn noch keine Zahl in ihm gespeichert wurde.

Wir können uns die ganze Situation wie bei einem Schrank mit vielen Schubladen vorstellen. Liest der Rechner `to PROG :PAR`, so öffnet er eine noch unbenutzte Schublade, beschriftet sie mit `PAR(PROG)` und legt eine 0 in ihr ab. Wird anschliessend `PROG 17` aufgerufen, so wird die Schublade mit dem Namen `PAR(PROG)` geöffnet, die 0 herausgenommen und eine 17 hineingelegt. Hiernach wird `PROG` ausgeführt und bei jedem Vorkommen von `:PAR` wird die entsprechende Schublade geöffnet und der Wert nachgeschaut und eingesetzt.



Wenn wir uns den Speicher des Rechners als Tabelle vorstellen, in der die einzelnen Zellen Register sind, können wir die Situation vor und nach der Ausführung von `QUADRAT 35` wie folgt darstellen.



Allgemein werden für ein Programm so viele Register „reserviert“, wie dieses Parameter in der „**to**-Zeile“ benutzt. Die Situation nach

```
to TEST :A :B :C :X :Y :Z
```

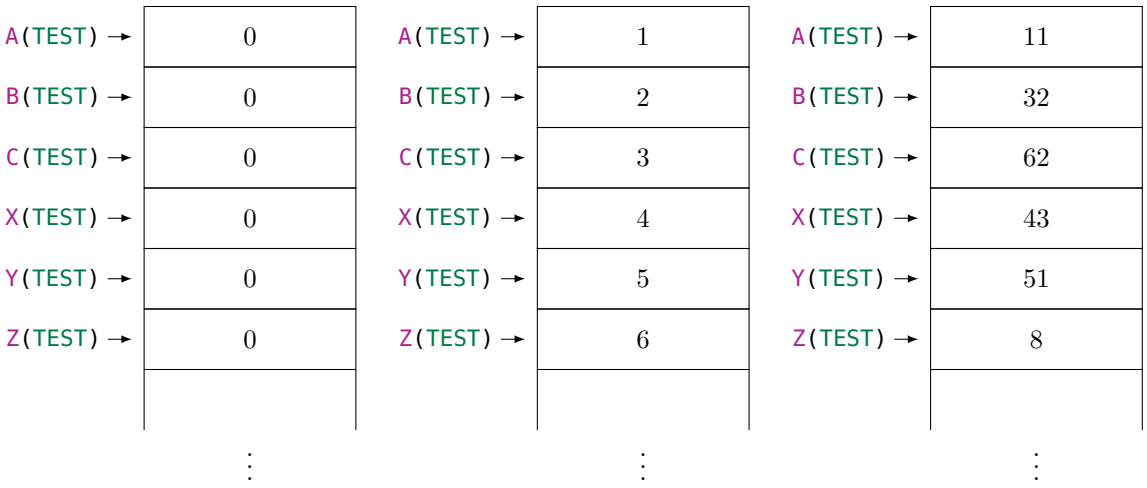
ist also wie in der folgenden Abbildung links dargestellt. Die Ausführungen von

```
TEST 1 2 3 4 5 6
```

beziehungsweise

```
TEST 11 32 62 43 51 8
```

führen dann zu Situationen wie in den Abbildungen in der Mitte beziehungsweise rechts.



Variablen

Wir haben bereits gelernt, dass Parameter Variablen sind, deren Werte sich während der Laufzeit nicht ändern. Es ist naheliegend, dass der Rechner für das Speichern aktueller Werte von Variablen ebenfalls Register verwendet. Anders als bei Parametern, die einen einmal gesetzten Wert während der gesamten Ausführung des Programms behalten, kann sich also der Inhalt eines Registers, das den aktuellen Wert einer Variablen repräsentiert, mit der Zeit ändern.

Betrachten wir beispielsweise das folgende Programm `VECKE`, in dem die Variable `:X` verwendet wird, um verschiedene Vielecke zu zeichnen.

```
    to VECKE
1.  make "X 100
2.  repeat 4 [fd :X rt 90]
3.  make "X :X+20
4.  repeat 6 [fd :X rt 60]
5.  make "X 1.1*:X+35
6.  repeat 3 [fd :X rt 120]
    end
```

Wir können nun den Inhalt des angelegten Registers `X(VECKE)` dokumentieren, indem wir notieren, wie er sich über die Zeit ändert, wenn `VECKE` aufgerufen wird. In der folgenden Tabelle halten wir in der i-ten Spalte fest, welchen Wert `X(VECKE)` nach der Durchführung der i-ten Zeile von `VECKE` enthält. Die 0-te Spalte entspricht dem Zeitpunkt nach dem Drücken der Eingabe-Taste, bevor der erste Befehl des Programms ausgeführt wurde. Wir deuten hier mit einem Strich an, dass der Wert der Variablen `:X` zu diesem Zeitpunkt noch nicht definiert ist.

	0	1	2	3	4	5	6
`X(VECKE)`	–	100	100	120	120	167	167

Wäre das Programm definiert mit `to VECKE :X`, hätten wir in der 0-ten Spalte die Zahl eingetragen, die dem Programm als Wert für `:X` übergeben wurde.

Aufgabe 20

Betrachte das folgende Programm.

```
    to VARSXY
1.  make "X 1
2.  make "X :X+15
3.  make "Y :X-13
4.  make "X :Y+4+2*:X
5.  make "Y :X
6.  make "X :Y
7.  make "X :Y+6
    end
```

Fülle die folgende Tabelle aus, indem du die Werte der Register `X(VARSXY)` und `Y(VARSXY)` für jede Zeile einträgst.

	0	1	2	3	4	5	6	7
`X(VARSXY)`								
`Y(VARSXY)`								

Aufgabe 21

Betrachte das folgende Programm.

```
    to SW
1.  make "A 9
2.  make "B 13
3.  make "A :A+:B
4.  make "B :A-:B
5.  make "A :A-:B
    end
```

Erkläre zunächst in Worten, was SW tut.

Fülle die folgende Tabelle aus, indem du die Werte der Register A(SW) und B(SW) für jede Zeile einträgst.

	0	1	2	3	4	5
A(SW)						
B(SW)						

Der Befehl `print`

Wir können mit Logo nicht nur Bilder zeichnen, sondern auch Texte ausgeben. Hierfür verwenden wir den **print**-Befehl (oder kurz **pr**), wobei der auszugebende Text in eckigen Klammern steht. Beispielsweise geben wir mit

```
pr [Hallo Welt!]
```

den Text „Hallo Welt!“ in der Nachrichten-Box aus. Besonders interessant ist **pr** für uns, da wir auch direkt den aktuellen Wert einer Variablen oder eines Parameters ausgeben können. In diesem Fall werden allerdings keine eckigen Klammern verwendet und wir können immer nur eine einzelne Variable benutzen. Somit führt

```
make "VAR 16
pr :VAR
```

zu der Ausgabe der Zahl 16. Ferner können wir den Befehl auch als Teil von komplexeren Folgen von Befehlen verwenden. Führe das Programm

```
to ZAEHLEN
make "X 0
repeat 100 [make "X :X+1 pr :X]
end
```

aus und überprüfe, ob das gewünschte Ergebnis ausgegeben wird.

Mit **pr** können wir nun zum Beispiel für das weiter oben dargestellte Programm VECKE nachvollziehen, wie sich der Inhalt des Registers X(VECKE) ändert, wenn es ausgeführt wird.

```
to VECKE
make "X 100
print :X
repeat 4 [fd :X rt 90]
make "X :X+20
print :X
repeat 6 [fd :X rt 60]
make "X 1.1*:X+35
print :X
repeat 3 [fd :X rt 120]
end
```

Die Ausführung führt dann in der Nachrichtenbox zu folgender Ausgabe.

```
100
120
167
```

Dies hilft uns, zu überprüfen, ob ein Programm so arbeitet, wie wir es wünschen.

Aufgabe 22

Erstelle ein Programm RECHTKALK mit zwei Parametern :HOR und :VER, das ein Rechteck der Grösse :HOR×:VER zeichnet und die Fläche und den Umfang dieses Rechtecks ausgibt.

Unterprogramme

Wir haben in Kapitel 6 bereits das Konzept von Unterprogrammen und der Übergabe von Parametern an Unterprogramme besprochen. Nun wollen wir untersuchen, was im Speicher des Rechners passiert, wenn Variablen an Unterprogramme übergeben werden.

Betrachten wir hierzu ein Programm QUADTREPPE2, das das bereits bekannte Programm QUADRAT benutzt, um eine leicht gedrehte Treppe von Quadraten zu zeichnen. Hierzu wird QUADRAT 3-mal mit jeweils verschiedenen Werten aufgerufen. Diese Werte werden durch die Parameter :G1, :G2 und :G3 des Hauptprogramms bestimmt.

```
    to QUADTREPPE2 :G1 :G2 :G3
1.  lt 5
2.  QUADRAT :G1
3.  pu rt 90 fd :G1+5 lt 90 pd
4.  QUADRAT :G2
5.  pu rt 90 fd :G2+5 lt 90 pd
6.  QUADRAT :G3
    end
```

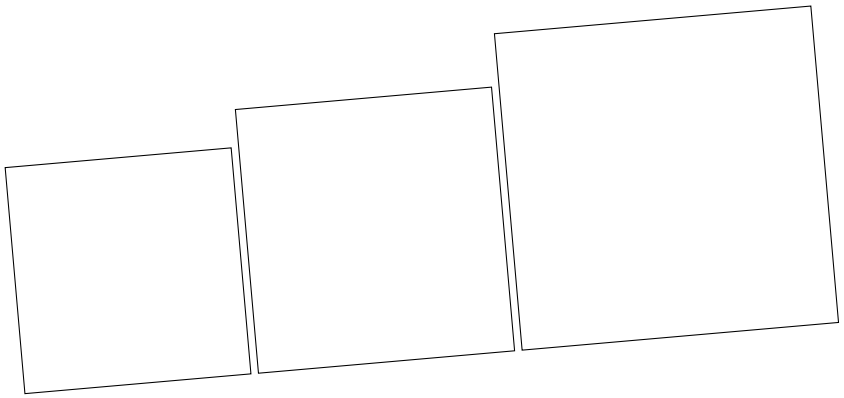
In diesem Beispiel ist also QUADRAT ein Unterprogramm von QUADTREPPE2. Das Unterprogramm verfügt über einen Parameter :GR, und diesem wird bei den drei Aufrufen jeweils ein anderer Wert übergeben. Alle Variablen, die in der durch **to** eingeleiteten Zeile vorkommen oder durch **make** in dem Programm definiert werden, nennen wir **globale Variablen**, da sie Variablen des Hauptprogramms QUADTREPPE2 sind. Die Variablen der Unterprogramme, hier also :GR, nennen wir **lokale Variablen** des Hauptprogramms QUADTREPPE2. Ferner ist :GR eine globale Variable des Programms QUADRAT.

Wir stellen fest, dass die Bezeichnung der Register so gewählt wird, dass in den Klammern immer das eindeutige Programm steht, für das die entsprechende Variable global ist.

Führen wir jetzt beispielsweise

```
QUADTREPPE2 100 110 135
```

aus, so wird folgendes Bild gezeichnet.



Nach der Definition der beiden Programme werden die Register G1(QUADTREPPE2), G2(QUADTREPPE2), G3(QUADTREPPE2) und GR(QUADRAT) reserviert und jeweils eine 0 in ihnen gespeichert. Wird dann QUADTREPPE2 wie oben aufgerufen, so werden die Werte 100, 110 und 135 in den entsprechenden Registern gespeichert, das Register GR(QUADRAT) wird zunächst jedoch nicht verändert, sondern erst, wenn QUADRAT als Unterprogramm aufgerufen wird.

Wir können wieder eine Tabelle ausfüllen, wobei wir in die i-te Spalte die Variablenwerte eintragen, die nach der Durchführung von Zeile i des Hauptprogramms in den entsprechenden Registern gespeichert sind.*

	0	1	2	3	4	5	6
G1(QUADTEST2)	100	100	100	100	100	100	100
G2(QUADTEST2)	110	110	110	110	110	110	110
G3(QUADTEST2)	135	135	135	135	135	135	135
GR(QUADRAT)	0	0	100	100	110	110	135

Wir können das Programm QUADTREPPE2 wie folgt zu einem Programm QUADTREPPE3 umschreiben, für das wir die drei Seitenlängen nicht mehr explizit angeben.

```
    to QUADTREPPE3 :START :DIFF1 :DIFF2
1.  lt 5
2.  make "G1 :START
3.  QUADRAT :G1
4.  pu rt 90 fd :G1+5 lt 90 pd
5.  make "G2 :G1+:DIFF1
6.  QUADRAT :G2
7.  pu rt 90 fd :G2+5 lt 90 pd
8.  make "G3 :G2+:DIFF2
9.  QUADRAT :G3
    end
```

*Es sei an dieser Stelle der Vollständigkeit halber angemerkt, dass wir hier die Sichtweise etwas vereinfacht haben. Das Programm QUADRAT wird dreimal aufgerufen und wir nehmen an, da dies nacheinander passiert, dass für alle Ausführungen dasselbe Register GR(QUADRAT) verwendet wird. Streng genommen unterscheidet der Rechner allerdings zwischen diesen verschiedenen Aufrufen von QUADRAT und legt mehrere Register an. Dies ist für uns an dieser Stelle allerdings vernachlässigbar.

Wir können nun folgende Klassifizierung vornehmen.

- `:START`, `:DIFF1`, `:DIFF2`, `:G1`, `:G2` und `:G3` sind globale Variablen von `QUADTREPPE3`.
- Davon sind `:START`, `:DIFF1`, `:DIFF2` globale Parameter von `QUADTREPPE3`, denn sie werden während der Laufzeit nicht verändert.
- `:GR` ist eine globale Variable (und ein globaler Parameter) von `QUADRAT`.
- `:GR` ist eine lokale Variable von `QUADTREPPE3`.

Aufgabe 23

Fülle die folgende Tabelle mit den Werten aus, die gespeichert werden, wenn `QUADTREPPE3 100 20 45` ausgeführt wird.

	0	1	2	3	4	5	6	7	8	9
START(QUADTREPPE3)										
DIFF1(QUADTREPPE3)										
DIFF2(QUADTREPPE3)										
G1(QUADTREPPE3)										
G2(QUADTREPPE3)										
G3(QUADTREPPE3)										
GR(QUADRAT)										

Aufgabe 24

In **??** haben wir das Programm `QUADTREPPE` mit zwei Parametern `:GROESSE` und `:STUFEN` geschrieben. Dieses benutzt als Unterprogramm wieder `QUADRAT`, das einen Parameter `:GR` besitzt. Fülle die folgende Tabelle mit den Werten aus, die gespeichert werden, wenn `QUADTREPPE` mit dem Wert 140 für `:GROESSE` und 5 für `:STUFEN` ausgeführt wird.

Hinweis: Anders als in dem Beispiel oben hast du für `QUADTREPPE` eine **`repeat`**-Schleife verwendet. Um die Werte der Register richtig zu dokumentieren, sollst du jede einzelne Ausführung der Schleife als einzelne Zeile des Programms beziehungsweise als einzelnen Zeitschritt betrachten.

	0	1	2	3	4	5	6	7
`GROESSE(QUADTREPPE)`								
`STUFEN(QUADTREPPE)`								
`GR(QUADRAT)`								

Aufgabe 25

Betrachte die folgenden beiden Programme, wobei EINKREIS ein Unterprogramm von DREIKREISE ist.

```
to EINKREIS :FARBE
setpc :FARBE
repeat 360 [fd 1 rt 1]
setpc 0
end
```

```
    to DREIKREISE :F1 :F2
1.  EINKREIS :F1
2.  rt 180
3.  EINKREIS :F2
4.  lt 90
5.  EINKREIS :F1
    end
```

DREIKREISE besitzt also zwei globale Parameter und einen lokalen. Fülle jetzt die unten stehende Tabelle für die Ausführung von DREIKREISE 2 4 aus. Die Spalten der Tabelle entsprechen hierbei den Zeitpunkten nach dem Ausführen der entsprechenden Zeilen des Hauptprogramms.

	0	1	2	3	4	5
F1(DREIKREISE)						
F2(DREIKREISE)						
FARBE(EINKREIS)						

Bislang hatten die Variablen in Programmen und Unterprogrammen stets verschiedene Namen, auch, wenn sie eigentlich dieselbe Funktion hatten. Wir haben zum Beispiel die Parameter, die die Seitenlängen eines Quadrates angeben, im Programm QUADTREPPE2 mit :G1, :G2 und :G3 bezeichnet, aber im Unterprogramm QUADRAT mit :GR. In der zweiten Zeile von QUADTREPPE2 rufen wir das Unterprogramm QUADRAT mit dem Parameter :G1 auf, der ein globaler Parameter von QUADTREPPE2 ist.

Das ist zunächst einmal unintuitiv und es erscheint uns sinnvoller, Variablen, die in verschiedenen Programmen für dasselbe stehen, auch gleich zu benennen. Aber „verwirren“ wir damit den Rechner nicht? Was würde konkret passieren, wenn wir QUADRAT wie folgt definiert hätten?

```
to QUADRAT :G1
repeat 4 [fd :G1 rt 90]
end
```

Dann wäre nach unserer Definition `:G1` einmal eine globale und einmal eine lokale Variable des Programms `QUADTREPPE2`.

Dieser scheinbare Konflikt führt für den Rechner allerdings zu keinerlei Problemen und wir wissen auch schon, wieso nicht. Für die globale Variable `:G1` von `QUADTREPPE2` wird ein Register `G1(QUADTREPPE2)` angelegt und für die globale Variable `:G1` von `QUADRAT` ein Register `G1(QUADRAT)`. Wird

```
QUADTREPPE2 90 130 150
```

ausgeführt, können wir den Speicher wie folgt darstellen.

	0	1	2	3	4	5	6
G1(QUADTREPPE2)	90	90	90	90	90	90	90
G2(QUADTREPPE2)	130	130	130	130	130	130	130
G3(QUADTREPPE2)	150	150	150	150	150	150	150
G1(QUADRAT)	0	0	90	90	130	130	150

Aufgabe 26

Ändere `QUADRAT` wie oben beschrieben ab und teste `QUADTREPPE2` mit verschiedenen Werten, um sicherzustellen, dass es noch immer das gewünschte Resultat erzielt.

Im obigen Beispiel wurden Variablen mit gleichem Namen in den Unterprogrammen nicht verändert. Wenn wir uns aber davon überzeugen wollen, dass der Rechner tatsächlich verschiedene Register benutzt, können wir dies mit den folgenden Programmen testen, die den Befehl **`print`** verwenden.

```
to HAUPTP
make "X 1
pr [Im Hauptprogramm, Wert:]
pr :X
UNTERP :X
pr [Wieder im Hauptprogramm, Wert:]
pr :X
end
```

```
to UNTERP :X
make "X 2
print [Im Unterprogramm, Wert:]
print :X
end
```

Führe das Programm HAUPTP aus. Anschliessend solltest du die folgende Ausgabe in der Nachrichtenbox sehen.

```
Im Hauptprogramm, Wert:
1
Im Unterprogramm, Wert:
2
Wieder im Hauptprogramm, Wert:
1
```

Dies ist genau das erwartete Verhalten.[†] Zunächst wird in der ersten Zeile von HAUPTP eine 1 im Register X(HAUPTP) gespeichert und dieser Wert dann ausgegeben. Dann wird das Unterprogramm UNTERP aufgerufen und im Register X(UNTERP) wird der Wert gespeichert, der im Register X(HAUPTP) steht, zunächst also eine 1. Danach wird der Wert im Register X(UNTERP) mit 2 überschrieben und ausgegeben. Anschliessend wird UNTERP beendet. Der in X(HAUPTP) gespeicherte Wert wurde nicht verändert und ist deswegen bei der Ausgabe noch immer 1.

[†]Wir müssen allerdings sicherstellen, dass in der **to**-Zeile von UNTERP angegeben wird, dass es eine Variable :X besitzt. Tun wir dies nicht, kann es zu unerwünschten Effekten kommen. Deswegen gilt es in diesem zweiten Fall, gleiche Variablennamen unbedingt zu vermeiden.

10 Verzweigungen und Schleifen

Bislang haben wir Programme geschrieben, die über Parameter gesteuert werden können und beispielsweise Bilder verschiedener Grösse zeichnen, je nachdem, wie der Wert eines Parameters gewählt wird. Oftmals ist es aber so, dass wir nur gewisse Werte für diese Parameter zulassen wollen. Es könnte zum Beispiel sein, dass ein Quadrat einen Höchstwert für die Seitenlänge besitzen soll, weil es sonst nicht mehr auf den Bildschirm passt. Falls der Wert des Parameters `:GR` diesen Wert überschreitet, soll der Rechner eine Fehlermeldung ausgeben und kein Quadrat zeichnen. Somit können wir sicherstellen, dass nur „sinnvolle“ Parameterwerte benutzt werden.

In diesem Kapitel lernen wir, wie Programme geschrieben werden können, die sich abhängig von Variablenwerten „verzweigen“, womit wir genau dieses Ziel erreichen können.

Der Befehl `if`

Mit dem **`if`**-Befehl können wir ein Programm schreiben, das sich je nachdem, ob eine bestimmte Bedingung erfüllt ist, auf eine gewisse Weise verhält oder nicht. Hierfür benutzen wir konkret die folgende Schreibweise.

`if` Bedingung **`[`** Tätigkeit 1 **`]`** **`[`** Tätigkeit 2 **`]`**

Ist die Bedingung erfüllt, so wird „Tätigkeit 1“ ausgeführt. Ist sie nicht erfüllt, so wird hingegen „Tätigkeit 2“ ausgeführt. Tätigkeiten können beliebige Abfolgen von Befehlen und Programmen sein.

Zum Beispiel könnte die Bedingung an den Wert einer Variablen gebunden sein. Wenn diese Variable einen gewissen Wert besitzt, so soll ein blaues Quadrat gezeichnet werden, sonst ein roter Kreis. Wir können dies mit einem Parameter `:WAS` dann wie folgt realisieren.

```
if :WAS=1 [setpc 4 repeat 4 [fd 50 rt 90]]
          [setpc 1 repeat 360 [fd 1 rt 1]]
```

Diesen Parameter können wir an ein Programm übergeben, das obige Zeilen beinhaltet.

Aufgabe 27

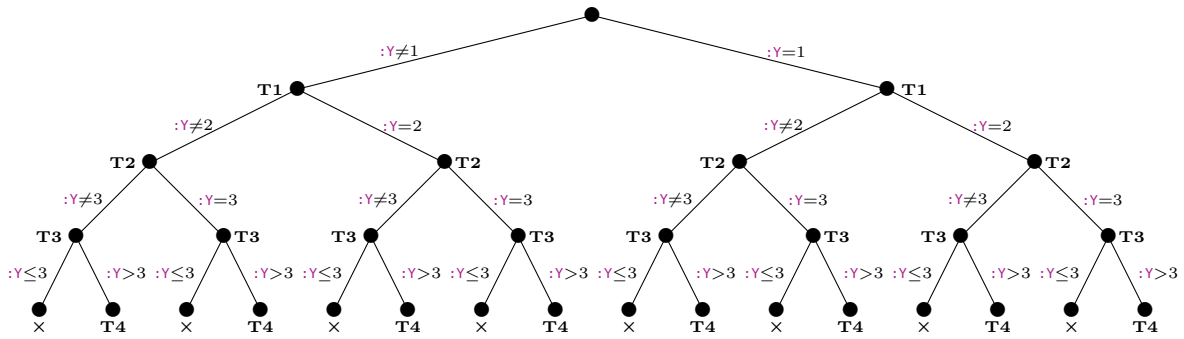
Schreibe ein Programm `WAHL`, das einen Parameter `:WAS` übergeben bekommt. Wenn `:WAS` den Wert 1 besitzt, so soll ein grünes Quadrat mit Seitenlänge 85 gezeichnet werden, sonst soll ein schwarzes Dreieck mit Seitenlänge 55 gezeichnet werden.

Der zweite Alternativ-Teil des **`if`**-Befehls (der Teil für „Tätigkeit 2“) kann auch leer gelassen werden. Wenn dies der Fall ist, so wird sofort mit dem Programm fortgefahren, wenn die Bedingung nicht erfüllt ist. Somit können wir also beispielsweise ein Programm `WAHL2` schreiben, das folgende Form besitzt.

```
to WAHL2
if :Y=1 [ Tätigkeit 1 ] []
if :Y=2 [ Tätigkeit 2 ] []
if :Y=3 [ Tätigkeit 3 ] []
if :Y>3 [ Tätigkeit 4 ] []
end
```

Die leeren Klammern `[` und `]` am Ende müssen nicht geschrieben werden, wir tun dies der Übersicht halber hier aber trotzdem.

Wir sprechen bei der Verwendung des **`if`**-Befehls auch von einer **Verzweigung** eines Programms. Woher diese Bezeichnung kommt, wird schnell klar, wenn wir folgende Darstellungsart wählen, die repräsentiert, wie der Lauf des Programms `WAHL2` in Abhängigkeit des Werts von `:Y` aussehen kann. Hierbei haben wir die Tätigkeiten mit „T“ abgekürzt, also beispielsweise „T1“ anstatt „Tätigkeit 1“ geschrieben. Ein Kreuz bedeutet, dass keine Tätigkeit ausgeführt wird.

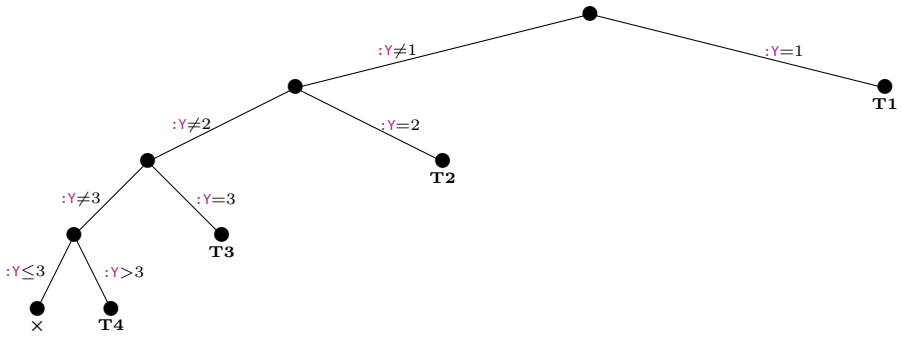


Wir nennen eine solche Darstellung ein **Baum-Diagramm**. Wenn wir genauer hinsehen, so stellen wir fest, dass dieses Baum-Diagramm sehr viele „Zweige“ beinhaltet, die nicht erreicht werden können, da `:Y` nicht gleichzeitig mehrere Werte besitzen kann. Offensichtlich ist es unnötig, zu überprüfen, ob `:Y` gleich 2 ist, wenn schon festgestellt wurde, dass `:Y` gleich 1 ist.

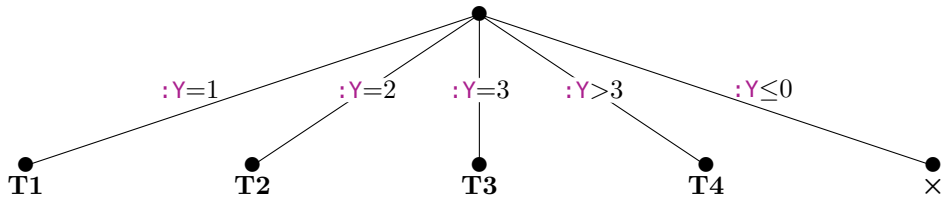
Im diesem Zusammenhang ist der Befehl **stop** von grossem Nutzen. Mit ihm wird das aktuelle Programm sofort beendet.

```
to WAHL2
if :Y=1 [ Tätigkeit 1 stop] []
if :Y=2 [ Tätigkeit 2 stop] []
if :Y=3 [ Tätigkeit 3 stop] []
if :Y>3 [ Tätigkeit 4 stop] []
end
```

Hiermit kürzen wir das Baum-Diagramm so, dass nur noch das Wesentliche dargestellt wird.



Wenn wir das Beispiel genau betrachten, so stellen wir fest, dass sich die verschiedenen Fälle, die in den Bedingungen des **if**-Befehls überprüft werden (`:Y=1`, `:Y=2`, `:Y=3` und `:Y>3`) gegenseitig ausschliessen. Eine intuitivere schematische Darstellung des Programms könnten wir deshalb wie folgt angeben.

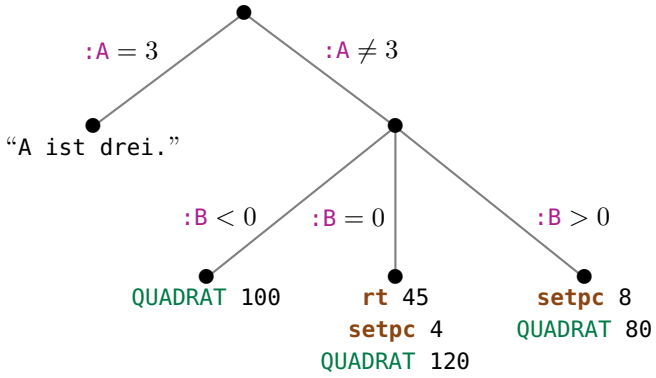


Untersuchen wir ein weiteres Beispiel.

```
if :A=3 [pr [A ist drei.] stop] []
if :B<0 [QUADRAT 100 stop] []
if :B=0 [rt 45 setpc 4 QUADRAT 120 stop] []
if :B>0 [setpc 8 QUADRAT 80 stop] []
```

Zunächst wird hier getestet, ob der Wert der Variablen `:A` gleich 3 ist. Falls ja, wird ein Text ausgegeben und das Programm beendet. Ansonsten wird der Wert der Variablen `:B` überprüft und abhängig davon, ob er negativ, positiv oder gleich 0 ist, ein anderes Quadrat gezeichnet. Wir können dies mit einem Diagramm ähnlich wie oben darstellen,

welches angibt, wie das Programm, abhängig von den Werten der Variablen :A und :B, „verzweigt“.



Aufgabe 28

Wie würde die Abbildung aussehen, wenn der Befehl **stop** in der ersten Zeile nicht verwendet worden wäre? Zeichne das entsprechende schematische Baum-Diagramm.

Um das Problem von Zeichnungen, die zu gross werden können, wieder aufzugreifen, betrachten wir nun das folgende Programm GEPRQUADRAT.

```
to GEPRQUADRAT :GR
if :GR>600 [pr [Die Seitenlänge ist zu gross.] stop] []
repeat 4 [fd :GR rt 90]
end
```

Hier wird als erstes überprüft, ob der Wert des Parameters :GR grösser ist als 600. Ist dies der Fall, wird das Programm mit einem Hinweis beendet, sonst wird ein Quadrat der angegebenen Seitenlänge gezeichnet.

Aufgabe 29

Schreibe das Programm GEPRQUADRAT so um, dass der Befehl **stop** nicht verwendet wird.

Um mehrere Bedingungen in einem Programm zu überprüfen, können wir **if**-Befehle „verschachteln“.

```
to SCHACHTEL :X :Y
if :X=1 [if :Y=1 [pr [Beide gleich eins.]]
                 [pr [Nur X gleich eins.]]]
        [pr [X nicht gleich eins.]]
end
```

Hier wird zunächst überprüft, ob der Wert von :X gleich 1 ist. Nur falls dies zutrifft, wird ebenfalls überprüft, ob der Wert von :Y gleich 1 ist.

Aufgabe 30

Schreibe ein Programm WAHL4 mit zwei Parametern :WAS und :WERT. Wenn der Wert von :WAS gleich 1 ist, soll ein blaues Quadrat mit einem Umfang von :WERT gezeichnet werden, aber nur, wenn der Wert von :WERT zwischen 200 und 1200 ist. Wenn der Wert von :WAS gleich 2 ist, so soll ein grüner Kreis mit Umfang :WERT gezeichnet werden, aber nur, wenn der Wert von :WERT grösser als 100 ist.

Benutze hierbei die Programme QUADUM und KREISUM als Unterprogramme.

Der Befehl mod

Wir haben schon gelernt, wie wir in Logo unter Verwendung der vier Grundrechenarten $+$, $-$, $\cdot$ und $/$ rechnen können. Jetzt wollen wir uns besonders mit der letzten Art, der Division, beschäftigen. Genauer geht es um den **Rest** der Division. Wenn wir beispielsweise 10 durch 3 teilen wollen, so ist dies nicht „glatt“ möglich. Die grösste Zahl kleiner als 10, die wir durch 3 teilen können, ist die 9. Da 9 um 1 kleiner ist als 10, sagen wir auch, dass 3 die Zahl 10 „mit Rest 1“ teilt. Genauso teilt 3 die 11 „mit Rest 2“. 12 hingegen wird von 3 „mit Rest 0“ beziehungsweise „ohne Rest“ geteilt.

Die wichtige Beobachtung für uns ist, dass eine natürliche Zahl durch eine andere natürliche Zahl teilbar ist, wenn der Rest dieser Division gleich 0 ist.

Mit dem Befehl

```
mod :X :Y
```

wird :X durch :Y geteilt und der Rest dieser Division zurückgegeben. Diesen können wir dann in einer Variablen speichern. Somit setzt

```
make "Z mod 10 3
```

den Wert von :Z auf 1. Probiere es aus, indem du den Wert anschliessend mit **pr** ausgeben lässt. Offensichtlich ist der Wert von :X genau dann durch den von :Y teilbar, wenn :Z anschliessend den Wert 0 besitzt.

Folglich können wir **mod** benutzen, um zum Beispiel zu testen, ob eine gegebene Zahl gerade, also durch 2 teilbar, ist. Betrachte hierzu das folgende Programm GERADEZAHL.

```
to GERADEZAHL :ZAHL
make "REST mod :ZAHL 2
if :REST = 0 [pr [Gerade.]]
             [pr [Ungerade.]]
end
```

Probiere es aus.

Wir haben in **??** in Kapitel 8 schon angemerkt, dass die Klammerung bei dem Befehl **sqrt** wichtig und ein bisschen unintuitiv ist. So verhält es sich bei allen Befehlen, mit denen wir rechnen können, also auch bei **mod**. Wenn wir den Rest der Division von 27 durch 8 zu 1 addieren wollen, müssen wir

```
(mod 27 8)+1
```

schreiben. Würden wir

```
mod 27 8+1
```

schreiben, so würde der Rechner zunächst 8+1=9 rechnen und anschliessend den Rest der Division von 27 durch 9 ausgeben.

Aufgabe 31

Entwirf ein Programm TEILBARKEIT4UND5, das einen Parameter :ZAHL erhält. Wenn der Wert von :ZAHL durch 4 teilbar ist, soll eine 1 ausgegeben werden. Falls nicht, soll eine 2 ausgegeben werden, wenn der Wert von :ZAHL durch 5 teilbar ist, sonst eine 3.

Zeichne zu deinem Programm ein schematisches Baum-Diagramm, das die Verzweigungen darstellt.

Der Befehl `while`

Wir haben in Kapitel 2 die **repeat**-Schleife kennengelernt, mit der sich eine beliebige Folge von Befehlen und Programmen wiederholen lässt.

Mit der **while**-Schleife lernen wir nun eine Verallgemeinerung kennen, mit der wir ein Programm so lange wiederholen können, wie eine bestimmte Bedingung erfüllt ist. Wie auch bei dem zuvor erwähnten Befehl **if** lässt sich hiermit beispielsweise sicherstellen, dass nur „sinnvolle“ Werte für einen Parameter benutzt werden.

Wir schreiben

while [Bedingung] [Tätigkeit]

um auszudrücken, dass eine Tätigkeit wiederholt werden soll, solange eine Bedingung erfüllt ist. Hierbei ist zu beachten, dass (anders als bei **if**) eckige Klammern um die Bedingung stehen müssen. Wir können so ein Programm schreiben, das Quadrate mit aufsteigender Seitenlänge zeichnet, allerdings nur, solange diese Länge kleiner als 1000 ist.

```
to QUADRATE3 :GR
while [:GR<1000] [
  QUADRAT :GR
  make "GR :GR+10
]
end
```

Probiere es aus. Wir können die **while**-Schleife immer anstatt einer **repeat**-Schleife verwenden, allerdings wird ein Programm hierdurch manchmal komplizierter.

Aufgabe 32

Schreibe das Programm QUADRAT zu einem Programm QUADRATWH um, sodass anstatt einer **repeat**-Schleife eine **while**-Schleife verwendet wird.

Hinweis: „Kleiner gleich“ musst du als „<=“ schreiben.

Eine wichtige Eigenheit, die wir bei der Verwendung von **while**-Schleifen (und auch **repeat**-Schleifen) beachten müssen, ist die Wirkung des eben eingeführten Befehls **stop**. Wird dieser nämlich innerhalb einer Schleife verwendet, so wird das Programm nicht beendet, sondern nur die Schleife verlassen. Betrachte die folgenden beiden Programme.

```
to WHSCHLEIFE
make "X 1
while [:X<=10] [
  pr :X
  if :X=5 [stop]
  make "X :X+1
]
pr [Hinter der Schleife.]
end
```

```
to RPTSCHLEIFE
make "X 1
repeat 10 [
  pr :X
  if :X=5 [stop]
  make "X :X+1
]
pr [Hinter der Schleife.]
end
```

Werden diese ausgeführt, erzeugt dies die folgende Ausgabe.

```
1
2
3
4
5
Hinter der Schleife.
```

Wollen wir erreichen, dass nicht nur die Schleife, sondern das ganze Programm beendet wird, können wir anstatt **stop** den Befehl **stopall** verwenden.

```
to WHSCHLEIFE2
make "X 1
while [:X<=10] [
  pr :X
  if :X=5 [stopall]
  make "X :X+1
]
pr [Hinter der Schleife.]
end
```

```
to RPTSCHLEIFE2
make "X 1
repeat 10 [
  pr :X
  if :X=5 [stopall]
  make "X :X+1
]
pr [Hinter der Schleife.]
end
```

Werden diese beiden Programme ausgeführt, wird die folgende Ausgabe erzeugt.

```
1
2
3
4
5
```

Allerdings müssen wird bei der Verwendung von **stopall** etwas anderes beachten, wenn dieser Befehl in Unterprogrammen verwendet wird. Wird er ausgeführt, so wird nämlich nicht nur das Unterprogramm, sondern auch das Hauptprogramm beendet. Wir können als Beispiel die beiden folgenden Programme betrachten.

```
to HAUPTP2
pr [Beginn der Hauptprogramms.]
UNTERP2
pr [Ende der Hauptprogramms.]
end
```

```
to UNTERP2
stopall
end
```

Wird jetzt HAUPT2 augeführt, so ist die Ausgabe die folgende.

```
Beginn des Hauptprogramms.
```

Das folgende Beispiel zeigt, wie wir manchmal mit **while**-Schleifen komplexere Rechnungen einfach ausführen können.

Die **Fakultät** $n!$ einer natürlichen Zahl n ist definiert als

$$n! = 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n,$$

also die Multiplikation aller natürlichen Zahlen von 1 bis n.

Die Fakultät ist eine wichtige Grösse, wenn es um das Abzählen von gewissen Möglichkeiten geht. Nehmen wir an, wir besitzen n Kugeln, die alle unterschiedliche Farben besitzen und wir fragen uns, auf wie viele unterschiedliche Arten wir sie in einer Reihe anordnen können.

Wir können uns dann Folgendes überlegen. Für die erste Position, also ganz links, können wir eine der n Kugeln nehmen und somit haben wir hierfür

$$n$$

Möglichkeiten. Ist eine Kugel ausgesucht, müssen wir noch bestimmen, wie viele Möglichkeiten wir für die zweite Position haben. Da eine Kugel schon auf der ersten Position liegt, sind dies

$$n-1$$

Möglichkeiten. Wollen wir nun wissen, wie viele Möglichkeiten es für Kombinationen der ersten und zweiten Position gibt, so müssen diese Zahlen multipliziert werden, denn für jede Wahl der ersten Position gibt es $n-1$ Möglichkeiten für die zweite. Somit gibt es

$$n \cdot (n-1)$$

Möglichkeiten für die ersten beiden Positionen.

Auf diese Art und Weise können wir weiter überlegen und folgern, dass es

$$n \cdot (n-1) \cdot (n-2)$$

Möglichkeiten für die ersten drei Positionen gibt und schliesslich

$$n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$$

für alle n Positionen.

Um die Fakultät einer gegebenen Zahl zu berechnen, können wir ein Programm `FAKUL` schreiben, das folgendermassen aussieht.

```
to FAKUL :ZAHL
make "ERG :ZAHL
while [:ZAHL>1][
  make "ZAHL :ZAHL-1
  make "ERG :ERG*:ZAHL
]
pr :ERG
end
```

Natürlich ist das Programm `FAKUL` nur eine Möglichkeit, $n!$ auszurechnen.

Aufgabe 33

Ändert sich die Ausgabe von `FAKUL`, wenn wir in der Bedingung der **`while`**-Schleife nicht `:ZAHL>1`, sondern `:ZAHL>2` schreiben würden? Warum beziehungsweise warum nicht?

Aufgabe 34

Ändert sich die Ausgabe von `FAKUL`, wenn wir in der Bedingung der **`while`**-Schleife nicht `:ZAHL>1`, sondern `:ZAHL>0` schreiben würden? Warum beziehungsweise warum nicht?

Aufgabe 35

Wenn wir FAKUL genau betrachten, so wird :ERG zu Beginn auf :ZAHL gesetzt und dann immer wieder mit der nächstkleineren Zahl multipliziert. Schreibe nun ein Programm FAKUL2, das mit der Zeile **make** "ERG 1 beginnt und in umgekehrter Reihenfolge vorgeht.

Der Befehl output

Wir haben bislang die Ergebnisse einer Berechnung in Variablen gespeichert und entweder auf dem Bildschirm ausgegeben oder in einer Zeichnung umgesetzt. Eine dritte Möglichkeit besteht darin, diese Ergebnisse beziehungsweise beliebige Inhalte von Variablen zurückgeben zu lassen, so dass sie von anderen Programmen weiterverarbeitet werden können. Dies ist sinnvoll, um Programme übersichtlich zu gestalten. Hiermit können wir sehr konsequent kleine Programme schreiben, die gewisse „einfache“ Aufgaben erledigen und die wir dann zusammensetzen und als Unterprogramme verwenden, um komplexere Aufgaben zu bewältigen. Der Unterschied zu Unterprogrammen, die direkt etwas zeichnen, ist, dass das Ergebnis der Arbeit von Unterprogrammen, die einen Wert zurückgeben, im Hauptprogramm weiterverarbeitet werden kann.

Zu diesem Zweck wird der Befehl **output** verwendet, der das Programm, in dem er aufgerufen wird, sofort beendet und den Wert einer angegebenen Variablen zurückgibt. Wenn diese Variablen den Namen :X besitzt, so wird

```
output :X
```

geschrieben. Betrachten wir zum Beispiel das Programm FAKUL3, das genau wie FAKUL arbeitet, aber in der letzten Zeile den Wert der Variablen :ERG nicht mit **pr** ausgibt, sondern **output** benutzt, um ihn weiterzuverarbeiten.

```
to FAKUL3 :ZAHL
make "ERG :ZAHL
while [:ZAHL>1][
  make "ZAHL :ZAHL-1
  make "ERG :ERG*:ZAHL
]
```

```
output :ERG
end
```

Wir können dieses Programm nun in einem Hauptprogramm FAKULQUAD verwenden, um für eine Zahl n die Funktion

$$(n!)^2$$

zu berechnen. Zu diesem Zweck speichern wir den von FAKUL3 zurückgegebenen Wert in einer Variablen, mit der wir dann weiterrechnen. Dies wird durch die Befehlsfolge

```
make "FA FAKUL3 :ZAHL
```

erreicht. Der Rechner führt zunächst das Programm FAKUL3 mit dem Parameter :ZAHL aus. Wenn dieses seine Berechnungen beendet hat, wird das Ergebnis x (also der Wert von :ERG) zurückgegeben, so dass nun

```
make "FA x
```

ausgewertet wird und somit die globale Variable :FA von FAKULQUAD den Wert x zugewiesen bekommt, den FAKUL3 berechnet hat.

```
to FAKULQUAD :ZAHL
make "FA FAKUL3 :ZAHL
make "ERG :FA*:FA
pr :ERG
end
```

Bei der Verwendung des Befehls **output** gilt es, einige Besonderheiten zu beachten. Programme, bei denen wir ihn verwenden, können nicht mehr direkt ausgeführt werden. Beispielsweise erhalten wir die Fehlermeldung

```
I don't know what to do with 120 ?
```

wenn wir FAKUL3 5 in die Befehlszeile eingeben. Ferner werden wir die mit solchen Programmen berechneten Werte immer zunächst in einer Variablen speichern, mit der wir dann im entsprechenden Hauptprogramm weiterrechnen können.

Aufgabe 36

Das folgende Programm POT hat zwei Parameter :A und :B und gibt

$$:A^{:B}$$

zurück, wobei es nur eine **while**-Schleife und Multiplikation benutzt.

Vervollständige den folgenden Programmcode an den durch ... markierten Stellen und gib ihn in deinen Editor ein.

```
to POT :A :B
make "IT 1
make "ERG :A
while [:IT< ...] [
  make "ERG :ERG*:A
  make "IT :IT ...
]
output :ERG
end
```

Aufgabe 37

Schreibe ein Programm `ADDPOT`, das drei Parameter `:A`, `:B` und `:C` übergeben bekommt und

$$(:A + :B)^{:C}$$

zurückgibt, wobei `POT` als Unterprogramm benutzt werden soll.

Eine weitere wichtige Frage, die in Verbindung mit dem Abzählen von Objekten steht, ist, wie viele Möglichkeiten es gibt, aus einer Menge von n verschiedenen Kugeln genau k zu ziehen, wenn einmal gezogene Kugeln nicht zurückgelegt werden.

Beim ersten Ziehen gibt es wieder genau n Möglichkeiten, beim zweiten $n-1$ Möglichkeiten und somit

$$n \cdot (n-1)$$

Kombinationen für die ersten beiden Ziehungen. Mit ähnlichen Überlegungen wie oben kommen wir schliesslich auf genau

$$n \cdot (n-1) \cdot \ldots \cdot (n-k+1)$$

Reihenfolgen, in denen die k Kugeln gezogen werden können.

Wenn wir diese Anzahl unter Verwendung von Fakultäten darstellen wollen, erhalten wir durch Multiplikation mit $(n-k)!$ im Zähler und Nenner

$$\frac{n \cdot (n-1) \cdot \ldots \cdot (n-k+1)}{1} = \frac{n!}{(n-k)!}.$$

Wir sind an dieser Stelle noch nicht fertig, denn die Reihenfolge der gezogenen Kugeln soll ignoriert werden, weswegen wir diese Zahl noch durch die Anzahl aller Reihenfolgen teilen müssen, in der die k Kugeln gezogen werden können. Wir haben schon gesehen, dass dies genau $k!$ entspricht.

Die obige Formel gibt also einen Wert an, der $k!$-mal zu gross ist. Wenn wir ihn durch $k!$ teilen, erhalten wir genau die gesuchte Anzahl der Möglichkeiten, k verschiedene Kugeln aus n zu ziehen. Diese Anzahl heisst **Binomialkoeffizient** und wird abgekürzt mit

$$\binom{n}{k} = \frac{n!}{(n-k)! \cdot k!}.$$

Aufgabe 38

Schreibe ein Programm `BINOM`, das das Programm `FAKUL3` benutzt, um aus zwei übergebenen Parametern `:N` und `:K` den Binomialkoeffizienten

$$\binom{:N}{:K}$$

zu berechnen. Hierbei soll zunächst überprüft werden, ob beide Werte grösser als 0 sind und ob ausserdem der Wert von `:N` nicht kleiner ist als der von `:K`.

Aufgabe 39

In einer vereinfachten Version des Schweizer Lottos werden zufällig 6 Zahlen aus 42 gezogen. Berechne die Anzahl der Möglichkeiten hierfür und somit die Chance, „einen Sechser im Lotto zu haben“.

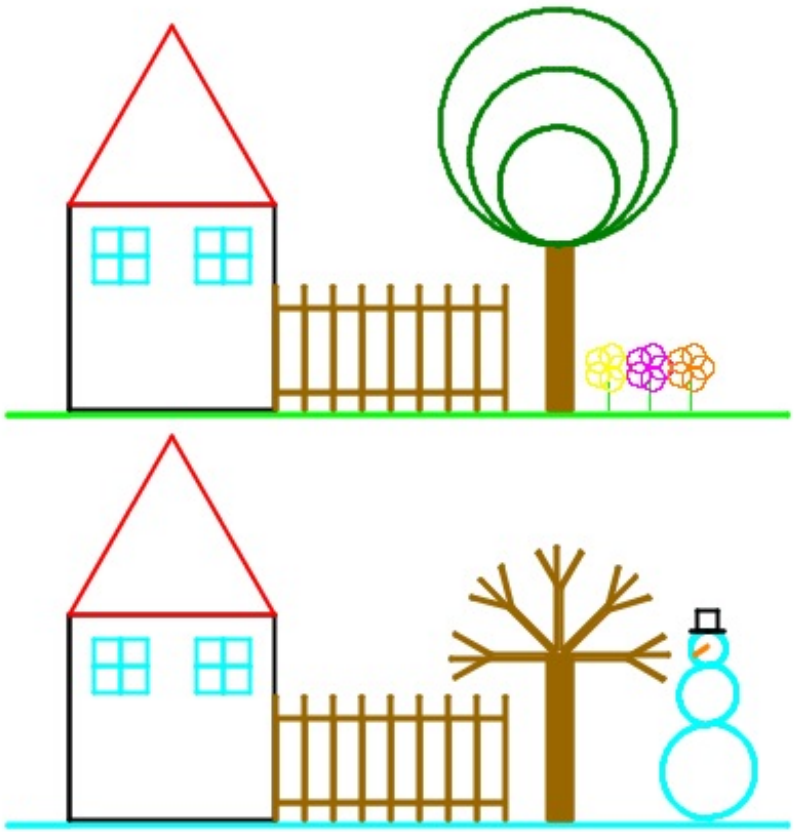
Hinweis: Da der Rechner hierfür mit sehr grossen Zahlen arbeiten muss, kann es sein, dass das Ergebnis etwas ungenau dargestellt wird und Kommastellen enthält.

Wir hoffen, dass dir das Bearbeiten der Unterlagen genau
so viel Spass gemacht hat wie uns das Erstellen.

Meine Notizen

Befehlsübersicht

`fd 100`	100 Schritte vorwärts gehen
`bk 50`	50 Schritte rückwärts gehen
`cs`	alles löschen und neu beginnen
`rt 90`	90 Grad nach rechts drehen
`lt 90`	90 Grad nach links drehen
`repeat 4 [...]`	das Programm in `[...]` wird viermal wiederholt
`pu`	die Schildkröte wechselt in den Wandermodus
`pd`	die Schildkröte wechselt zurück in den Stiftmodus
`setpc 3`	wechselt die Stiftfarbe auf die Farbe 3
`setpw 5`	wechselt den Stift-Durchmesser auf 5
`to NAME`	erstellt ein Programm mit einem Namen
`to NAME :PARAMETER`	erstellt ein Programm mit einem Namen und einem Parameter
`end`	alle Programme mit einem Namen enden mit diesem Befehl
`pe`	die Schildkröte wechselt in den Radiergummimodus
`ppt`	die Schildkröte wechselt zurück in den Stiftmodus
`wait 5`	die Schildkröte wartet 5 Zeiteinheiten
`make "VARIABLE 10`	setzt den Wert einer Variablen auf 10
`sqrt :VARIABLE`	berechnet die Quadratwurzel einer Variablen
`pr :VARIABLE`	gibt den aktuellen Wert einer Variablen aus
`if :X=1 [P1] [P2]`	führt `P1` aus, wenn `:X` gleich 1 ist und sonst `P2`
`stop`	beendet das aktuelle Programm oder die aktuelle Schleife
`stopall`	beendet alle Programme
`mod 13 5`	gibt den Rest bei Division von 13 durch 5, also 3, aus
`while [:VARIABLE>1] [...]`	das Programm in `[...]` wird wiederholt solange der Wert von `:VARIABLE` grösser 1 ist
`output :VARIABLE`	gibt den Wert von `:VARIABLE` zurück
`home`	setzt die Schildkröte wieder auf die Startposition



Programmieren mit LOGO
für Fortgeschrittene

Informationstechnologie und Ausbildung
ETH Zürich, CAB F 15.1
Universitätstrasse 6
CH-8092 Zürich

www.ite.ethz.ch
www.abz.inf.ethz.ch