

Programmieren Lernen

Timm Knappe

12. April 2020

1 Programmieren ist Zauberei

1.1 Im Keller

Das Gewölbe ist düster. Grauer Rauch steigt langsam wabernd aus großen, bronzenen Schalen auf. Von verborgenen LED-Strahlern wird er lila angestrahlt und mystifiziert die gesamte Szene. Leise klirren indische Klang-Schalen kreuz und quer, ohne sich verorten zu lassen.

Ein Mann in langem Mantel steht auf einmal mitten im Raum. Niemand hat ihn kommen sehen. Auf dem Mantel sind mit goldenen Fäden Symbole aus unterschiedlichen Schrift-Systemen aufgestickt. Wenn etwas nach einem bekannten lateinischen Buchstaben aussieht, dann ist es sicherlich ein griechisches, kyrillisches oder ganz anderes Symbol.

Der Mann breitet langsam die Arme aus. Gleichzeitig verändert sich der Raum. Die Temperatur sinkt. Das Licht nimmt sich zurück. Nur eine gelbe Licht-

säule bildet sich um den Magier.

Wo vorher noch unbestimmte Leere war, leuchtet nun ein Pentagramm auf dem groben Felsboden. In dessen Mitte steht der Zauberkundige.

Er murmelt kaum verständlich eine Beschwörungsformel in einer fremden Sprache. Teile erinnern an Latein. Dabei bewegen sich seine Finger als wären sie zu einem eigenen, unabhängigen Leben erwacht und erkunden wie frisch geschlüpfte Küken die große Welt.

Ruckartig senkt er die Arme. Ein Donnerschlag dröhnt durch die Halle. Ein kleiner Bistro-Tisch steht wie aus dem Nichts vor ihm. Auf diesem liegen dampfende Papp-Kartons.

„Bitte nehmen Sie sich, Ihre Pizza. Ich hoffe, der Tag hat Ihnen gefallen.“

Mit einer knappen Verbeugung verabschiedet sich der Künstler und verläßt das Gewölbe, in dem eine Management-Schulung zu ihrem Abschluß gekommen ist.

1.2 Quer-Bezug

Ja und? Was hat das alles mit Programmieren zu tun? Eine Menge!

Für den Außenstehenden wirkt Programmieren wie Magie. Programmierkundige können seelenlose Maschinen zum Leben erwecken. Durch die zunehmende Vernetzung werden die Programme scheinbar allwissend.

Wie ein böser Dämon müssen sie von ihrem Meister gebändigt und in Zaum gehalten werden.

Aber gleichzeitig ist das Programmieren für erfahrene Programmierer eben keine Hexerei. So wie auch ein Zauberkünstler keine echte Magie braucht, um seine Kunststücke vorzuführen.

Dieses Buch ist der erste Schritt ei-

ner Anleitung zum Programmieren.

Alles was du brauchst ist ein Gerät mit Internet-Anschluß, auf dem ein Web-Browser läuft. Zum Beispiel ein Tablet oder Smart-Phone. Oder ein Laptop. Oder ein Desktop-Computer. Das sind diese Kisten mit abgesetzten Bildschirm und Tastatur.

Damit können wir ein klein wenig programmieren lernen. Es ist gar nicht so schwer. Die Programme sind unsere Zaubersprüche. Der Web-Browser ist der Raum, in dem wir wirken. Und mit etwas Geduld entstehen komplexe Gebilde, die scheinbar viel mächtiger sind, als die paar Zeilen Programm-Text vermuten lassen.

Lass uns das Spiel beginnen!

2 Um was geht es?

Es ist total wichtig, dass wir ganz klar Wissen, um was es überhaupt geht. Wenn wir kein gemeinsames Verständnis der verwendeten Begriffe haben, dann ist es höchstens Glück, wenn Wissen und Erkenntnis transportiert werden. Das kann funktionieren. Muss es aber nicht.

2.1 Definitionen

Zuerst geht es um unsere eigene Rolle. Der *Programmierer* oder die *Programmiererin* erstellen Programme. Gute Programmierer zeichnen sich dadurch aus, dass sie recht schnell Pro-

gramme erstellen können, die relativ wenig Fehler haben und selber schnell laufen. Aber das sind vorerst nur Details. Wichtig ist: Wir müssen Programme schreiben.

Programme selbst sind Anweisungen, die so klar und haarklein umrissen sind, dass selbst eine Maschine sie ausführen kann. Es gibt unterschiedliche *Programmiersprachen*, in denen Programme formuliert werden können.

Ein Beispiel ist JavaScript, das heute in fast jedem Web-Browser verwendet werden kann. Aber JavaScript hat so seine Tücken. Es wird leichter sein, mit einer einfacheren Sprache anzufangen.

Als drittes Element gibt es noch die *Maschine*, welche ein Programm ausführt. Das kann ein Computer sein. Muss es aber nicht.

Das folgenden Beispiel zeigt, wie man schon um 1900 ein Programm ganz ohne Elektronik schreiben und ausführen konnte.

2.2 Beispiel: Taylorismus

Die Fabriken sind ein schönes Beispiel für das nicht-elektronische Ausführen eines Programms. Man spricht auch gerne vom Abarbeiten eines Programms: Es gibt eine Liste von Schritten, die nacheinander ausgeführt werden müssen.

Zur Ehre des großen Pionier-Geists von Henry Ford und der Besinnung als Teil einer Auto-Nation, habe ich ein etwas vereinfachtes Programm geschrieben, wie ein Auto in einer Fabrik gebaut wird:

1. Nehme vier Reifen r_1, \dots, r_4 .
2. Nehme ein Lenkrad rr .
3. Baue in wenigen, nicht näher beschriebenen Schritten aus r_1, \dots, r_4, rr mit zusätzlichem Material einen roten VW Polo.

Natürlich war die eigentliche Liste in Wolfsburg etwas länger. Aber die würde den Rahmen sprengen und rechtliche Streitigkeiten heraufbeschwören.

Bleiben wir bei den drei Schritten.

Die Programmierer in der Fabrik sind die Ingenieure und Wissenschaftler,

die alle Schritte zusammentragen, die notwendig sind, um ein Auto zu bauen.

Je genauer die Schritte beschrieben sind, desto einheitlicher sind die resultierenden Autos. Und desto weniger muss der Fließband-Arbeiter in der Fabrik vom Auto-Bauen verstehen.

In unserem vereinfachten Programm sind die ersten beiden Schritte mit einigen Minuten anlernen ausführbar: Da hinten liegen Reifen, dort im Regal sind die Lenkräder. Nun sieh zu!

Für den dritten Schritt braucht es etwas mehr Expertise. Die ich leider selber nicht besitze. Daher muss ich beim Auto-Bauen notgedrungen eine sehr kompetente Maschine zum Ausführen meines Programms voraussetzen.

In diesem Beispiel ist die Fabrik-Halle mit ihren Arbeitern, Fließ-Bändern und Lackier-Robotern die Maschine, die das Programm „ich baue einen Polo“ ausführen kann.

Dazu benötigt die Fabrik zusätzliches Material als Eingabe. Irgendwo müssen auch die Räder und Lenkräder herkommen. Auch Betriebsmittel wie Strom und Geld sind notwendig.

Und sie produziert ein Auto als Ausgabe. Diese Begriffe werden wir später noch präzisieren müssen.

2.3 Etwas realistischer

Ein Programm zum Auto-Bauen ist heute gar nicht mehr so abwegig.

Heute können Autos vielfältig konfiguriert werden. Das erleichtert zum

einen den Händlern, sich um das Rückgaberecht zu drücken. Aber auch die Kunden genießen, dass ihr Auto ganz individuell zu ihnen passt und nicht in einem Einheits-Schwarz wie alle anderen Autos herumfährt. Obwohl schwarz immer noch eine sehr verbreitete Farbe ist.

Aber die Konfiguration eines Autos ist im Prinzip auch ein Programm. Es bekommt nicht jeder Arbeiter in der Fabrik eine Kopie meines Bestell-Zettels, aber er bekommt eine Liste mit Schritten, die er ausführen muss, um genau mein Auto zu bauen.

Diese Listen werden nicht händisch erstellt. Vielmehr gibt es ein Programm, das aus der Konfiguration (die ja wie gesagt auch ein Programm ist) ein anderes Programm macht. Solche Programme nennt man *Compiler*. Und mit ihnen kann man jede Menge Schabernack anstellen.

2.4 Andere Namen

Zusammen mit dem Programm wird oft der Begriff *Algorithmus* verwendet. Ein Algorithmus beschreibt, wie ein Programm funktioniert. Er ist meistens nicht in einer Programmier-Sprache geschrieben, sondern abstrakt. Ein Computer kann einen Algorithmus nicht direkt ausführen. Ein Mensch kann es jedoch. Also ist ein Algorithmus durchaus ein Programm für die Maschine

Programmierer. Meistens übernimmt es dann der Programmierer den Algorithmus in ein Programm einer anderen Programmiersprache zu übersetzen, so dass ein Computer ihn ausführen kann.

Aber für uns macht das erst einmal keinen Unterschied. Ein Algorithmus ist ein Programm für eine bestimmte Maschine (uns!). Ein Algorithmus hat noch zusätzliche Einschränkungen, die an dieser Stelle noch nicht behandelt werden sollen.

Gerne wird anstatt des Begriffs Programm auch der *Prozess* verwendet. Besonders wenn die ausführende Maschine Menschen enthält. Aber auch handelt es sich nur um ein Programm für eine bestimmte Maschine. Ähnlich wie beim Algorithmus sind die Unterschiede hauptsächlich ästhetischer Natur.

Koch-Rezepte werden auch immer wieder gerne als ein Beispiel für Programme herangezogen. Dem kann ich nur anschließen. Unser Begriff des Programms ist allgemein genug, um Rezepte mit zu umfassen. Am Beispiel des Rezeptes können auch wieder schön die einzelnen Komponenten unterschieden werden. Ein Rezept macht noch keinen Eierpfannkuchen. Dazu benötigt man noch eine ausführende Maschine (den Koch) und die notwendigen Zutaten (Eier, Mehl) und Betriebsmittel (Herd, Pfanne). Nur so kann die erwünschte Ausgabe produziert und danach verzehrt werden.

3 Zeichnen lassen

Genug der Vorrede. Auf zum ersten richtigen Programm! Unter <https://itmm.github.io/yoshi/> gibt es eine Web-Seite mit zwei Feldern. In das eine Feld kann das Programm eingegeben werden. Das Programm besteht aus Mal-Anweisungen.

Die Anweisungen richten sich an die Schildkröte Yoshi aus der Mario-Welt, die in der Mitte des zweiten Feldes sitzt

und nach oben sieht. Ihre Aufgabe ist es, Fahrbahnmarkierungen auf eine neue Straße zu zeichnen. Aber ihr muss ganz genau gesagt werden, was sie zeichnen muss. Aus lizenz-rechtlichen Gründen wird Yoshi selber nicht gezeichnet.

Nach Klick auf den „Auftrag ausführen“-Knopf werden die Anweisungen ausgeführt. Das Ergebnis erscheint im anderen Feld:

Yoshi zeichnet Straßen

Yoshis Auftrag:

```
(markiere 20)
(drehe 120)
(markiere 20)
(drehe 120)
(markiere 20)
(drehe 120)
```

Ergebnis:



Auftrag ausführen

Sehen wir uns das Programm genauer an:

```
1 (markiere 20)
2 (drehe 120)
3 (markiere 20)
4 (drehe 120)
5 (markiere 20)
6 (drehe 120)
```

Jede Zeile ist eine eigene Anweisung. Jede Anweisung beginnt mit `(` und endet mit `)`.

Das erste Wort in der Anweisung ist der `*Name*` der Anweisung. Er sagt Yoshi, was für eine Aktion er ausführen soll. Im ersten Programm gibt es nur die Namen `markiere` und `drehe`.

Die erste Anweisung (markiere 20) fordert die Schildkröte auf, zwanzig Schritte in die aktuelle Richtung zu laufen. Dabei hinterläßt sie eine Linie.

Die nächste Anweisung (drehe 120) dreht die Schildkröte um 120 Grad (eine Drittel-Drehung) im Uhrzeigersinn. Sie blickt nun nach rechts/unten. Die nächste Linie fährt also in einem spitzen Winkel in diese Richtung.

Nach insgesamt drei Markierungen und Drehungen steht Yoshi wieder auf seinem Startpunkt und blickt wieder nach oben. Zusätzlich hat sie aber ein Dreieck gezeichnet.

3.1 Ein Quadrat zeichnen

Wie sieht nun ein Programm aus, dass Yoshi dazu bringt ein Quadrat zu zeich-

nen?


Anstatt drei Linien müssen vier Linien gezogen werden. Und anstatt von Drittel-Drehungen müssen Viertel-Drehungen ausgeführt werden.

Das Programm kann wie folgt aussehen:

- 1 (markiere 20)
- 2 (drehe 90)
- 3 (markiere 20)
- 4 (drehe 90)
- 5 (markiere 20)
- 6 (drehe 90)
- 7 (markiere 20)
- 8 (drehe 90)

Es liefert das folgende Ergebnis:

Yoshi zeichnet Straßen

Yoshis Auftrag:	Ergebnis:
<pre>(markiere 20) (drehe 90) (markiere 20) (drehe 90) (markiere 20) (drehe 90) (markiere 20) (drehe 90)</pre>	
<div>Auftrag ausführen</div>	

3.2 Erste Erkenntnisse

Jetzt haben wir schon zwei Programme geschrieben: Mit dem ersten malt Yoshi ein Dreieck. Mit dem zweiten malt die Schildkröte ein Quadrat. Schon beim Quadrat fällt auf, wie mühevoll es ist, Yoshi geometrische Formen zu erklären. Zwar kann man mit etwas probieren Programme schreiben, die nun auch Vierecke, Fünfecke und so weiter schreiben. Aber für die Errechnung der Winkel wird irgendwann ein Taschenrechner notwendig. Und komfortabel ist die Eingabe auch nicht.

Ich gebe zu, ich habe beim Ausprobieren nur die ersten zwei Zeilen angepasst, diese kopiert und dann weitere Male eingefügt. Die restlichen Zeilen des alten Programms habe ich gelöscht. Wenn Programmiersprachen keine Wiederholungen unterstützen, ist das leider das Einzige, was bleibt.

Das hat gravierende Nachteile: Wenn man später etwas verändern will, müssen viele Zeilen angepasst werden. Wenn man ein paar vergißt, funktioniert das Programm vielleicht nicht mehr richtig. Bereits bei der Umschreibung vom Dreiecks-Programm zum Quadrat-Programm kann man dies beobachten.

Man sollte meinen, dass es in den letzten Jahren genug Fortschritte gegeben hat, um uns diese Mühsal zu ersparen. Leider weit gefehlt: das Kopieren und wiederholte Einfügen ist zum Beispiel bei Tabellenkalkulationen immer noch der Weg der Wahl.

Zum Glück ist Yoshi cleverer.

3.3 Wiederholen

Das Programm zum Zeichnen eines Quadrats kann man auch so aufschreiben:

```
1 (wiederhole 4
2   (markiere 20)
3   (drehe 90)
4 )
```

Das ist viel weniger zum Tippen, aber bringt das gleiche Resultat. Programmierer sind faul: warum arbeiten, wenn der Rechner auch die Arbeit für einen erledigen kann. Oftmals ist der Computer dabei gründlicher und macht weniger Fehler.

Aber erst einmal soll geklärt werden, was das Programm überhaupt macht, bzw. wieso es funktioniert.

Der erste Befehl heißt `wiederhole` und hat drei Argumente. Ja, genau: 3. Das erste Argument ist eine Zahl. Die beiden weiteren Argumente sind wieder Befehle!

Der `wiederhole` Befehl nimmt das erste Argument und führt die weiteren Argumente so oft aus, wie in dem ersten Argument angegeben wurde.

Oder noch genauer: solange das erste Argument größer als 0 ist, werden die weiteren Befehle ausgeführt und danach das erste Argument um 1 reduziert. Anstatt 4 könnte man im Programm auch 3.2 schreiben. Wichtig ist, dass Dezimalzahlen wie im Englischen mit einem Dezimal-Punkt anstatt des

deutschen Dezimal-Kommas geschrieben werden müssen. Aber grundsätzlich kann jede Zahl als erstes Argument von `wiederhole` verwendet werden.

Bleibt nur das lästige Problem mit der Winkel-Berechnung.

3.4 Winkel berechnen

Das können wir zum Glück auch dem Rechner überlassen.

Yoshi soll nach dem Ausführen in die gleiche Richtung blicken. Also muss er sich um `360` Grad (oder ein Vielfaches davon) drehen.

Beim Quadrat muss er sich also um $360/4 = 90$ Grad drehen, da ein Quadrat vier Seiten hat.

Beim Dreieck muss er sich um $360/3 = 120$ Grad drehen.

Die Berechnung kann auch Yoshi ausführen. Aber auf eine etwas komische Art. Es gibt einen Befehl, der eine Zahl durch eine andere teilt. Wie in der obigen Formel heißt dieser Befehl `/`.

Aber der Befehl muss ja immer als erstes Element der Liste stehen. Um also $360/4$ zu berechnen, lautet der Befehl `(/ 360 4)`. Diesen können wir das Quadrat-Programm einsetzen:

```
1 (wiederhole 4
2   (markiere 20)
3   (drehe (/ 360 4))
4 )
```

3.5 Geschachtelte Befehle

Die Befehle `wiederhole` und `drehe` gehen mit ihren Argumenten unterschied-

lich um.

Beim Befehl `drehe` (und auch bei vielen anderen Befehlen) kann man annehmen, dass geschachtelte Befehle ausgeführt werden, *bevor* der eigentliche Befehl ausgeführt wird.

Der Befehl `drehe` sieht also keine Division als Argument, sondern nur das Ergebnis: die Zahl 90. So funktioniert es bei fast allen Befehlen, bis auf ein paar Ausnahmen. Zu diesen *Spezial-Formen* gehört auch `wiederhole`.

`wiederhole` sieht die übergebenen Befehle und führt sie so oft aus, wie nötig ist.

Woran kann man Befehle von Spezial-Formen unterscheiden? Leider gibt es keine klare Regel. Wenn ein Befehl nicht als Spezial-Form benannt wird, dann wird es sich hoffentlich um einen normalen Befehl handeln.

3.6 Fünfeck und Pentagramm

Wenn nun ein Fünfeck gezeichnet werden soll, müssen nur noch die beiden `4` durch `5` ersetzt werden. Zusätzlich wird die Länge `20` etwas reduziert. Sonst passt das Fünfeck nicht mehr in das Feld.

Das resultierende Programm sieht so aus:

```
1 (wiederhole 5
2   (markiere 15)
3   (drehe (/ 360 5))
4 )
```

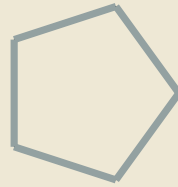
Das Programm liefert das folgende Ergebnis:

Yoshi zeichnet Straßen

Yoshis Auftrag:

```
(wiederhole 5  
  (markiere 15)  
  (drehe (/ 360 5))  
)
```

Ergebnis:



Auftrag ausführen

Vorher habe ich geschrieben, dass sich Yoshi auch um ein Vielfaches von 360° drehen kann. Probieren wir das aus, indem sich Yoshi zweimal um 360° dreht.

Angenommen, wir wissen nicht, dass $2 \cdot 360 = 720$ ist. Dann können wir auch Yoshi wieder mit der Aufgabe betrauen.

Hier ist das neue Programm:

```
1 (wiederhole 5  
2   (markiere 20)  
3   (drehe (/ (* 2 360) 5))  
4 )
```

Als Ergebnis erhalten wir ein Pentagon:

Yoshi zeichnet Straßen

Yoshis Auftrag:

```
(wiederhole 5
  (markiere 20)
  (drehe (/ (* 2 360) 5))
)
```

Ergebnis:



Auftrag ausführen

3.7 Aufgabe 1: Modernes Dreieck

Wie kann das ursprüngliche Programm

```
1 (markiere 20)
2 (drehe 120)
3 (markiere 20)
4 (drehe 120)
5 (markiere 20)
6 (drehe 120)
```

mit **wiederholung** und Division vereinfacht werden?

3.8 Aufgabe 2: Innenwinkel

Der *Innenwinkel* bei einem gleichseitigen Dreieck beträgt 60° , nicht 120° . Auch das folgende Programm zeichnet ein Dreieck, dreht sich aber nur um 60° .

```
1 (markiere -20)
2 (drehe 60)
3 (markiere 20)
4 (drehe 60)
5 (markiere -20)
6 (drehe 60)
```

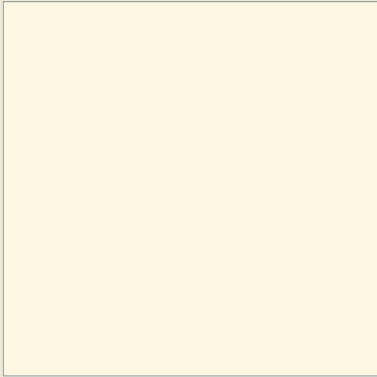
1. Wie unterscheiden sich die gezeichneten Dreiecke?
2. Welche Nachteile hat das Programm (z.B. Möglichkeiten der Vereinfachung, Orientierung von Yoshi am Ende)?

3.9 Aufgabe 3: Rose

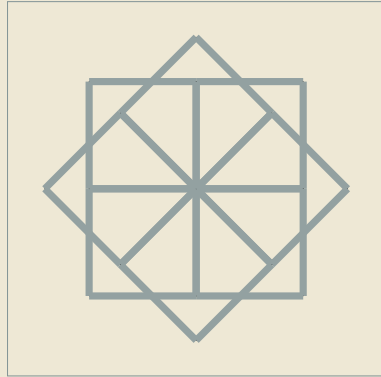
Welches Programm liefert das folgende Ergebnis?

Yoshi zeichnet Straßen

Yoshis Auftrag:



Ergebnis:



Auftrag ausführen

3.10 Aufgabe 4: Kreiselei

1. Was passiert, wenn sich Yoshi im Fünfeck-Programm dreimal oder viermal um 360° dreht?
2. Gibt es ein Muster?
3. Wie sieht dieses Muster beim Dreieck und beim Quadrat aus?

- 1 (markiere 20)
- 2 (drehe 90°)
- 3 (markiere 10)
- 4 (drehe 90°)
- 5 (markiere 20)
- 6 (drehe 90°)
- 7 (markiere 10)
- 8 (drehe 90°)

1. Wie kann es mit `wiederhole` vereinfacht werden?

3.11 Aufgabe 5: Rechteck

Das folgende Programm zeichnet ein Rechteck:

2. Zeichne die Rose aus Aufgabe 3 mit einem Rechteck anstelle einem Quadrat.

4 Sachen benennen

Bisher können wir schon recht komplexe Grafiken mit wenigen Zeilen Code zeichnen lassen. Die Aufgabe 4: Rose

gibt ein schönes Beispiel.

Jedoch gibt es bei unserem Polyeder Programm noch eine unschöne Wieder-

holung:

```
1 (wiederhole 5
2   (markiere 15)
3   (drehe (/ 360 5))
4 )
```

Die Zahl `5` muss an zwei Stellen eingegeben werden. Einmal um anzugeben, wie viele Seiten der Polyeder haben soll. Und einmal um den korrekten Winkel zu berechnen.

Wie sähe ein allgemeines Programm aus, um einen Polyeder zu zeichnen?

Das folgende Programm definiert eine Funktion `poly` und verwendet sie danach wie die eingebauten Befehle `markiere`, `drehe` oder `+`:

```
1 (def-fn poly (n)
2   (wiederhole n
3     (markiere 15)
4     (drehe (/ 360 n))
5   )
6 )
7 (poly 5)
```

Der Funktion wird ein Parameter `n` übergeben. Innerhalb des Funktionsaufrufs wird `n` dann durch den konkreten Wert `5` ersetzt, mit dem die Funktion aufgerufen wurde.

Allgemein definiert die Spezial-Form `def-fn` eine neue Funktion. Der Name der Funktion ist das erste Argument. Eine Liste mit Argumenten der Funktion ist das zweite Argument. Alle weiteren Argumente werden beim Aufruf der Funktion ausgeführt.

Yoshi verwendet eine *funktionale Programmiersprache*, die an die Pro-

grammiersprache LISP angelehnt ist. Daher werden Funktionen in der weiteren Betrachtung einen wichtigen Teil einnehmen.

Auch wenn LISP schon ein paar Tage auf dem Buckel hat (die Sprache wurde bereits 1958 spezifiziert), sind die Prinzipien heute aktueller denn je. Langsam fangen sie an, der objektorientierten Programmierung den Rang abzulassen. Das zeigen zum Beispiel aktuelle Erweiterungen der Sprachen Java und C++, aber auch moderne Sprachen wie Haskell, Scala oder Clojure.

4.1 Funktionsaufruf sezieren

Betrachten wir Schritt für Schritt, was bei einem Methoden-Aufruf passiert. Dies sind nicht genau die gleichen Schritte, die ein Rechner durchführt, aber sie sind einfacher zu erklären und führen zum gleichen Ergebnis.

Nehmen wir den Aufruf `(poly 5)`. Man kann den Aufruf durch die Kommandos der Funktion ersetzen. Dabei muss das Argument `n` immer durch den Wert `5` ersetzt werden. Damit ergibt sich:

```
1 (wiederhole 5
2   (markiere 15)
3   (drehe (/ 360 5))
4 )
```

Den `wiederhole` Befehl können wir ebenfalls auseinanderrollen und erhalten:

```
1 (markiere 15)
```

```
2 (drehe (/ 360 5))
3 (markiere 15)
4 (drehe (/ 360 5))
5 (markiere 15)
6 (drehe (/ 360 5))
7 (markiere 15)
8 (drehe (/ 360 5))
9 (markiere 15)
10 (drehe (/ 360 5))
```

Und wenn die Division ausgeführt wurde bleibt folgendes übrig:

```
1 (markiere 15)
2 (drehe 72)
3 (markiere 15)
4 (drehe 72)
5 (markiere 15)
6 (drehe 72)
7 (markiere 15)
8 (drehe 72)
9 (markiere 15)
10 (drehe 72)
```

Diese Instruktionen führt Yoshi aus und zeichnet das Fünfeck.

4.2 Mehrere Funktions-Argumente

Die Funktion `poly` hat nur ein einziges Argument: die Anzahl der Seiten des Polyeders. Aber es können auch mehrere sein.

Angenommen, wir möchten uns mit der Länge einer Seite ebenfalls nicht festlegen. Es hat sich bereits gezeigt, dass der Wert 20 nicht immer funktioniert und zum Beispiel manchmal durch 15 ersetzt werden muss. Bei Polyedern mit noch mehr Ecken muss die Länge noch kleiner gewählt werden.

Ein neues Programm kann so aussehen:

```
1 (def-fn poly (n l)
2   (wiederhole n
3     (markiere l)
4     (drehe (/ 360 n))
5   )
6 )
7 (poly 5 15)
8 (poly 5 10)
9 (poly 5 5)
```

Es entsteht folgendes Bild:

Yoshi zeichnet Straßen

Yoshis Auftrag:

```
(def-fn poly (n l)
  (wiederhole n
    (markiere l)
    (drehe (/ 360 n))
  )
)
(poly 5 15)
(poly 5 10)
(poly 5 5)
```

Ergebnis:



Auftrag ausführen

Hier sieht man einen zweiten Vorteil der Funktion: Eine Funktion zu schreiben und einmal aufzurufen, sieht nach zusätzlichem Aufwand ohne klaren Nutzen aus. Aber sobald die Funktion mehrfach aufgerufen wird, reduziert sich die Programmgröße erheblich.

Eine Funktion ist nicht nur ein Element, um Programme besser zu strukturieren, sondern reduziert bei richtiger Anwendung auch deren Größe und Komplexität.

Den Aufruf `(poly 5 10)` kann man sich wieder als eine sehr kompakte Schreibweise von

```
1 (wiederhole 5
2   (markiere 10)
3   (drehe (/ 360 5))
4 )
```

vorstellen. Mit den oben beschriebenen

weiteren Vereinfachungen.

Doch warum bei zwei aufhören. Mit einem weiteren Parameter `r` können wir angeben, wie oft sich Yoshi um die eigene Achse drehen soll:

```
1 (def-fn poly (n l r)
2   (wiederhole n
3     (markiere l)
4     (drehe (/ (* r 360) n))
5   )
6 )
7 (poly 5 15 2)
```

Mit jedem zusätzlichen Parameter wird die Funktion flexibler. Der Aufruf wird jedoch immer komplexer. Oft ist es schwierig, die richtige Anzahl an Parametern abzuwägen.

Vielleicht verwenden wir fast immer nur Polyeder mit einer einfachen Drehung, und davon haben die meisten eine

Kantenlänge von 20. Aber eben nicht immer.

Wir können natürlich mehrere Funktionen schreiben:

```
1 (def-fn poly (n l r)
2   (wiederhole n
3     (markiere l)
4     (drehe (/ (* r 360) n))
5   )
6 )
7 (def-fn poly-1 (n l)
8   (wiederhole n
9     (markiere l)
10    (drehe (/ 360 n))
11  )
12 )
13 (def-fn std-poly (n)
14   (wiederhole n
15     (markiere 20)
16     (drehe (/ 360 n))
17   )
18 )
19 (poly 4 10 1)
20 (poly-1 4 15)
21 (std-poly 4)
```

Noch einfacher wird das Programm, wenn man erkennt, dass die zweite und dritte Funktion nur ein Sonderfall der anderen ist:

```
1 (def-fn poly (n l r)
2   (wiederhole n
3     (markiere l)
4     (drehe (/ (* r 360) n))
5   )
6 )
7 (def-fn poly-1 (n l)
8   (poly n l 1)
9 )
10 (def-fn std-poly (n)
11   (poly-1 n 20)
12 )
13 (poly 4 10 1)
14 (poly-1 4 15)
15 (std-poly 4)
```

Betrachten wir den Aufruf von (std-poly 4) mit der Ersetzungsregel. Aus dem Aufruf wird:

```
1 (poly-1 4 20)
```

Das **n** von **poly-1** wurde durch **4** und das **l** durch **20** ersetzt. Die nächste Ersetzung ergibt:

```
1 (poly 4 20 1)
```

Das **n** von **poly** wurde durch **4**, das **l** durch **20** und **r** durch **1** ersetzt. Im nächsten Schritt ergibt sich:

```
1 (wiederhole 4
2   (markiere 20)
3   (drehe (/ (* 1 360) 4))
4 )
```

Das Ersetzen der Multiplikation ergibt:

```
1 (wiederhole 4
2   (markiere 20)
3   (drehe (/ 360 4))
4 )
```

Das Ausrechnen der Division ergibt:

```
1 (wiederhole 4
2   (markiere 20)
3   (drehe 90)
4 )
```

Und mit dem Ersetzen der Wiederholung ergibt sich:

- 1 (markiere 20)
- 2 (drehe 90)
- 3 (markiere 20)
- 4 (drehe 90)
- 5 (markiere 20)
- 6 (drehe 90)
- 7 (markiere 20)
- 8 (drehe 90)

Nicht immer macht es Sinn, alle diese Schritte im Kopf durchzuführen. Aber gerade am Anfang hilft es unheimlich, um zu verstehen, wie ein Funktionsaufruf funktioniert.

4.3 Frage: Kann eine Funktion sich auch selber aufrufen?

Auf jeden Fall! Darin besteht die Mächtigkeit der funktionalen Programmierung.

Jedoch können wir einen solchen Aufruf bisher noch nicht verwenden. Wir haben noch keine Möglichkeit zu beschreiben, dass eine Funktion sich *nicht immer* aufruft.

Wenn sie sich aber immer aufruft, dann wird das Programm nie beendet.

Und da im Browser erst die Grafiken neu gezeichnet werden, nachdem das Programm beendet wurde, macht ein solches Programm keinen Sinn. Es handelt sich vielmehr um einen Programmierfehler.

4.4 Sichtbarkeit von Bezeichnern

Bisher sind die unterschiedlichsten Bezeichner vorgekommen. Ein paar Beispiele sind `markiere`, `wiederhole`, `+`, `n`.

Hinter einem Bezeichner versteckte sich entweder eine Funktion (bei `markiere` und `+`), eine Spezial-Form (bei `wiederhole`) oder eine Zahl (bei `n`).

Jedoch sind nicht alle Bezeichner immer gültig.

Betrachten wir das Programm

```
1 (def-fn poly (n)
2   (wiederhole n
3     (markiere 20)
4     (drehe (/ 360 n))
5   )
6 )
7 (poly 3)
```

Insgesamt, kommen folgende Bezeichner vor:

- `def-fn`, `wiederhole` (Spezial-Form),
- `poly`, `markiere`, `drehe` (Funktion),
- `n` (Argument/Zahl).

Bis auf `n` und `poly` können diese Bezeichner überall verwendet werden.

`poly` kann nur verwendet werden, nachdem die Funktion mit `def-fn` definiert wurde. Folgendes Programm macht keinen Sinn:


```

2 (def-fn poly (n)
3   (wiederhole n
4     (markiere 20)
5     (drehe (/ 360 n))
6   )
7 )

```

Bei der Ausführung kommt eine Fehlermeldung, das die Funktion `poly` nicht bekannt ist.

Die Verwendung von `n` ist noch eingeschränkter: es kann nur innerhalb der Definition der Funktion verwendet werden.

Erst bei einem Funktionsaufruf bekommt `n` einen konkreten Wert zugewiesen (z.B. `3` bei `(poly 3)`). Nur dann werden die Kommandos in der Funktion abgearbeitet und nur dann können sie `n` verwenden.

Nach dem Aufruf der Funktion ist `n` nicht mehr sichtbar und kann nicht mehr verwendet werden.

Man kann sich das mit einem Stapel Kisten vorstellen. Zu Beginn gibt es nur eine Kiste. In der sind alle Spezial-Formen und alle globalen Funktionen abgelegt.

Wenn mit `def-fn` eine neue Funktion definiert wird, dann wird diese Funktion in die oberste Kiste des Stapels abgelegt.

Um zu einem Bezeichner den entsprechenden Wert oder die passende Funktion zu finden, wird in der obersten Kiste nachgesehen. Gibt es dort einen Treffer, so wird er zurück geliefert. Andernfalls wird in der darunter liegenden Kiste nachgesehen. Und so weiter.

Bei einem Funktionsaufruf passiert nun etwas Komisches: Zuerst wird für jedes Argument der Wert ermittelt. Dafür wird der aktuelle Stapel zu Rate gezogen. Dann werden viele Kisten zur Seite gestellt, bis die oberste Kiste die Definition der Funktion enthält. Darauf wird eine neue Kiste gestellt. In diese Kiste kommt ein Eintrag für jedes Argument, welches für den Argument-Namen den beim Aufruf ermittelten Wert ablegt. Mit diesem Stapel werden die Kommandos der Funktion abgearbeitet.

Zum Ende des Funktionsaufrufs werden die neu angelegten Kisten wieder entfernt. Die oberste Kiste enthält nun wieder die Definition der Funktion. Die zur Seite gestellten Kisten kommen wieder oben auf den Stapel. Nun sieht der Stapel wieder so aus, wie beim Aufruf der Funktion.

Nehmen wir das folgende Programm:

```

1 (def-fn poly (n)
2   (wiederhole n
3     (markiere 15)
4     (drehe (/ 360 n))
5   )
6 )
7 (def-fn poly + 1 (n)
8   (poly (+ 1 n))
9 )
10 (poly + 1 4)

```

Die Gemeinheit besteht darin, dass `n` in zwei Funktionen als Argument verwendet wird. Und die eine Funktion die andere auch noch aufruft.

Es handelt sich jedoch um unter-

schiedliche Werte, da sie in unterschiedlichen Kisten liegen.

Aber unser Ersetzungs-Prinzip schafft hier Klarheit. Innerhalb von `poly + 1` hat `n` den Wert 4. Innerhalb von `poly` jedoch den Wert 5.

Betrachten wir den Aufruf `(poly + 1 4)` detailliert. Die erste Ersetzung ergibt:

```
1 (poly (+1 4))
```

und das wird zu `(poly 5)` aussummiert. Die nächste Ersetzung ergibt:

```
1 (wiederhole 5
2   (markiere 15)
3   (drehe (/ 360 5))
4 )
```

mit den üblichen Fortführungen.

4.5 Innere Funktionen

Funktionen können auch in einer Funktion definiert werden.

```
1 (def-fn poly (n)
2   (def-fn geh-dreh ()
3     (markiere 20)
4     (drehe (/ 360 n))
5   )
6   (wiederhole n
7     (geh-dreh)
8   )
9 )
10 (poly 3)
```

Nach dem Kisten-Prinzip ist die Funktion `geh-dreh` nur innerhalb eines Funktionsaufrufs von `poly` sicht-

bar. Sie kann daher im `wiederhole`-Aufruf verwendet werden.

Oft können innere Funktionen verwendet werden, um neue Bezeichner einzuführen. Eine andere Definition von `poly` wäre:

```
1 (def-fn poly (n)
2   (def-fn inner (w)
3     (wiederhole n
4       (markiere 20)
5       (drehe w)
6     )
7   )
8   (inner (/ 360 n))
9 )
10 (poly 3)
```

Dieses Programm ist komplizierter, als die ursprüngliche Version:

```
1 (def-fn poly (n)
2   (wiederhole n
3     (markiere 20)
4     (drehe (/ 360 n))
5   )
6 )
7 (poly 3)
```

Es wird statt dessen eine neue Funktion geschrieben und diese mit dem berechneten Winkel aufgerufen.

Diese Variante kann in manchen Situationen deutlich schneller sein. Anstatt dass für jede Seite neu in der `dreh`-Funktion der Winkel mit einer Division neu berechnet wird, muss die Division nur einmal durchgeführt werden.

Eine Division ist noch nicht so schlimm (es sein denn, die Anzahl `n`

der Seiten wird sehr, sehr groß), aber wenn statt dessen eine kompliziertere Berechnung ausgeführt wird, kann das ein echter Zeitgewinn werden.

4.6 Rückgabewerte

Jede Funktion gibt auch einen Wert zurück. Das ist der Wert der letzten Funktion, die innerhalb der Funktion ausgeführt wird.

Betrachten wird das folgende Programm:

```
1 (def-fn winkel (n)
2   (/ 360 n)
3 )
4 (def-fn poly (n)
5   (wiederhole n
6     (markiere 20)
7     (drehe (winkel n))
8   )
9 )
10 (poly 3)
```

Die Funktion `winkel` führt die Division aus und gibt den neuen Wert zurück.

Der Parameter von `winkel` kann vermieden werden, wenn wir die Funktion als innere Funktion verwenden:

```
1 (def-fn poly (n)
2   (def-fn winkel ()
3     (/ 360 n)
4   )
5   (wiederhole n
6     (markiere 20)
7     (drehe (winkel n))
8   )
9 )
10 (poly 3)
```

Nehmen wir gleich eine weitere Funktion für die Kantenlänge:

```
1 (def-fn poly (n)
2   (def-fn winkel ()
3     (/ 360 n)
4   )
5   (def-fn länge ()
6     20
7   )
8   (wiederhole n
9     (markiere (länge))
10    (drehe (winkel n))
11  )
12 )
13 (poly 3)
```

Wenn wir eine Kantenlänge l haben, so hat der Polyeder einen Umfang von $u = l \cdot n$. Wenn es sich um einen Kreis handeln würde, dann wäre der Durchmesser $d \cdot \pi = u$.

Für jeden Polyeder gilt aber immer noch $d \cdot \pi \geq l \cdot n$. Da d nicht über 20 wachsen soll, gilt für l : $l \leq 20 \cdot \pi/n$.

Das können wir einbauen:

```
1 (def-fn poly (n)
2   (def-fn winkel ()
3     (/ 360 n)
4   )
5   (def-fn länge ()
6     (/ (* 20 3.142) n)
7   )
8   (wiederhole n
9     (markiere (länge))
10    (drehe (winkel n))
11  )
12 )
13 (poly 7)
14 (drehe 180)
15 (poly 20)
```

Zur Optimierung kann der Aufruf von `winkel` und `länge` auch nur einmal erfolgen:

```
1 (def-fn poly (n)
2   (def-fn winkel ()
3     (/ 360 n)
4   )
5   (def-fn länge ()
6     (/ (* 20 3.142) n)
7   )
8   (def-fn inner (w l)
9     (wiederhole n
10      (markiere l)
11      (drehe w)
12    )
13  )
14  (inner (winkel) (länge))
15 )
16 (poly 7)
```

Durch die Abstraktion der Länge in eine eigene Funktion macht es auf einmal Sinn, sich mit dem Wert zu befassen und ihn zu optimieren.

4.7 Funktionen als Argumente

Jetzt kommen wir zum Höhepunkt dieses Heftes.

Funktionen können auch anderen Funktionen als Argumente übergeben oder auch als Rückgabewerte zurück gegeben werden.

Betrachte dazu folgendes Programm:

```
1 (def-fn rotate(n f)
2   (def-fn inner (w)
3     (wiederhole n
4       (f)
5       (drehe w)
6     )
7   )
8   (inner (/ 360 n))
9 )
10 (def-fn poly (n)
11   (def-fn mark ()
12     (markiere 20)
13   )
14   (rotate n mark)
15 )
16 (poly 3)
```