

Einführung in die funktionale Programmierung mit Lisp

Fabian Eberts

23. Oktober 2020

Inhaltsverzeichnis

| | |
|--|-----------|
| 1. Einleitung | 2 |
| 2. Die Sprache Scheme | 3 |
| 2.1. Scheme – ein Lisp-Dialekt | 3 |
| 2.2. Literatur | 3 |
| 2.3. Installation der Entwicklungsumgebung | 4 |
| 3. Einführung in Scheme | 5 |
| 3.1. Geklammerte Präfixnotation | 5 |
| 3.2. Arithmetische Ausdrücke | 6 |
| 3.3. Zeichenketten | 7 |
| 3.4. Funktionen und λ -Kalkül | 8 |
| 3.5. Kontrollstrukturen | 9 |
| 3.6. Listen und Paare | 9 |
| 3.7. Mapping | 11 |
| 3.8. Folding | 12 |
| 4. Bilderzeugung | 14 |
| 4.1. Funktionsgraphen | 14 |
| 4.2. Geometrische Objekte | 16 |
| 4.3. Fraktale | 17 |
| 5. Zusammenfassung | 19 |
| Literaturverzeichnis | 20 |

1. Einleitung

Dieses Skript stellt den Lisp-Dialekt *Scheme* einführend vor. Dabei ist es von Vorteil, wenn der Leser bereits über Erfahrungen in der funktionalen Programmierung verfügt und mit den wichtigsten Konzepten vertraut ist. Diese werden nur dann (kurz) erläutert, wenn dies sinnvoll erscheint oder es sich um Scheme-spezifische Besonderheiten handelt. Diese Einführung eignet sich also insbesondere für diejenigen, die eine weitere funktionale Sprache kennenlernen möchten. Aber auch Neulingen kann dieser Schnelleinstieg nützlich sein. Hier ist allerdings die Hinzunahme weiterführender Literatur ratsam. Entsprechende Empfehlungen werden in Abschnitt 2.2 ausgesprochen.

Motivation

Neben Lisp gibt es noch eine Menge anderer funktionaler Sprachen. Die Erfahrung zeigt, dass das Erlernen einer weiteren Sprache desselben Programmierparadigmas meist weniger Aufwand erfordert, weil man die wesentlichen Konzepte bereits verinnerlicht hat. So fällt beispielsweise der Einstieg in Java nicht schwer, wenn man zuvor schon mit C++ programmiert hat. Häufig lernt man auf diese Weise erst die Stärken und Schwächen bestimmter Sprachen kennen. Der „Blick über den Tellerrand“ lohnt sich allemal.

Ziele

Wir werden die Sprache Scheme genauer kennenlernen. Nach dem Durcharbeiten dieses Skripts werden wir in der Lage sein, kleine Programme selbst zu schreiben. Außerdem werden wir uns so weit an die Syntax gewöhnt haben, dass uns das selbstständige Erlernen weiterer Sprachelemente nicht schwerfallen wird.

Aufbau

In Kapitel 2 verschaffen wir uns zunächst einen kleinen Überblick über die Sprache Scheme und installieren einen Interpreter. Die Einführung in die Programmierung erfolgt dann in Kapitel 3. Zu guter Letzt werden wir uns in Kapitel 4 noch ein wenig Spaß gönnen und uns mit einer Grafikbibliothek beschäftigen, mit deren Hilfe wir Funktionsgraphen plotten und Fraktale erzeugen werden.

2. Die Sprache Scheme

In diesem Kapitel versuchen wir, Scheme in der Landschaft der vielen Lisp-Dialekte zu verorten. Dann wird die Literatur vorgestellt, welche für dieses Skript herangezogen wurde und auch darüber hinaus genutzt werden kann, um Scheme weiter zu erlernen. Anschließend machen wir uns mit der Entwicklungsumgebung *DrRacket* vertraut.

2.1. Scheme – ein Lisp-Dialekt

Scheme entstand 1975 als Lisp-Dialekt. Obwohl Scheme eine funktionale Sprache ist, unterstützt sie auch andere Programmierparadigmen. Aufgrund ihrer einfachen Syntax wird sie gerne in der Lehre eingesetzt. Trotz des minimalistischen Designs ist Scheme eine ausdrucksstarke Sprache und ermöglicht das Schreiben von Programmen hoher Komplexität. Der Kern der Sprache besteht aus wenigen grundlegenden Konstrukten, aus welchen sich die gesamte Sprache herleiten lässt. Das erklärt nicht nur ihre Flexibilität, sondern auch die große Anzahl verschiedener Dialekte. Heute ist Scheme eine Allzweck-Programmiersprache und kommt in der Forschung und Softwareentwicklung zum Einsatz [Dyb09, *Preface* und *1. Introduction*]. Laut Tiobe-Index sind Scheme und Haskell in ihrer Popularität etwa gleichauf. Erst deutlich danach kommt Common Lisp. Racket taucht im Index (noch) gar nicht auf [Tio20]. Die derzeit aktuellste Spezifikation von Scheme ist der R⁷RS (*Revised⁷ Report on the Algorithmic Language Scheme*) aus dem Jahr 2013 [SCG13].

2.2. Literatur

Im Folgenden sind einige Quellen sowie weiterführende Literatur aufgeführt:

- Als Hauptquelle für dieses Skript – insbesondere für Kapitel 3 – dient *The Scheme Programming Language* [Dyb09]. Es handelt sich dabei um ein sehr beliebtes Lehrwerk für die Sprache Scheme. Das Buch enthält viele Übungsaufgaben und ist als Onlineversion verfügbar.
- Ein anderes Standardwerk ist *Structure and Interpretation of Computer Programs* [ASS96]. Dabei geht es in erster Linie um das Erlernen grundlegender und fortgeschrittener Konzepte der Programmierung im Allgemeinen. Dabei wird Scheme verwendet, um die verschiedenen Themen anhand praktischer Beispiele zu erklären. Das Buch eignet sich eher als weiterführende Lektüre, nachdem man bereits Grundkenntnisse in Scheme erlangt hat. Es liegt eine Onlineversion vor.

- Empfehlenswert ist außerdem *The Little Schemer* [FF95]. Man könnte das Buch als eine eher unkonventionelle Einführung in Scheme beschreiben. Ein Schwerpunkt ist die rekursive Programmierung. Das Buch ist nicht frei im Internet verfügbar.

2.3. Installation der Entwicklungsumgebung

Eine bequeme Möglichkeit, erste Erfahrungen mit Scheme zu sammeln, bietet die Entwicklungsumgebung *DrRacket*, welche zusammen mit der Racket-Plattform installiert werden kann. Zwar dient diese primär zum Programmieren mit *Racket* (einem weiteren Dialekt von Lisp, welcher auf Scheme basiert), doch kann dort auch reiner Scheme-Code ausgeführt werden.








Auf der Website¹ wird ein Installer für gängige Linux-Distributionen angeboten. Nach dem Herunterladen wird die Plattform wie folgt installiert:

```
user@linux:~$ ls -l racket-7.7-x86_64-linux.sh
-rwxrwxr-x 1 user user 123001690 Jun 24 20:51 racket-7.7-x86_64-linux.sh
user@linux:~$ chmod +x racket-7.7-x86_64-linux.sh
user@linux:~$ ./racket-7.7-x86_64-linux.sh
```

Während der Installation werden verschiedene Optionen angeboten. Zum Experimentieren mit Scheme ist es dabei am praktischsten, die Plattform in ein eigenes Verzeichnis zu installieren. Anschließend wird DrRacket mit `./racket/bin/drracket` gestartet.

Die Umgebung ist in zwei Fenster unterteilt. Im oberen Fenster muss die zu verwendende Sprache eingegeben werden (hier `#lang scheme`) und dann der *Start*-Button gedrückt werden. Das untere Fenster ist der Interpreter, in welchem einzelne Befehle, aber auch ganze Programme ausgeführt werden können:

```
> "Hallo Scheme"
"Hallo Scheme"
> (* 6 7)
42
```

Ausgeführt wird ein Befehl mit der -Taste, sofern sich der Cursor am Zeilenende befindet. Tut er das nicht, drückt man stattdessen +. Ansonsten wird ein Zeilenumbruch eingefügt, denn es können auch mehrzeilige Befehle eingegeben werden. Durch die Historie kann man sich mit + und + bewegen.

Im nächsten Kapitel lernen wir Scheme kennen. Alle dort gezeigten Beispiele wurden im DrRacket-Interpreter getestet.

¹ <https://racket-lang.org/>

3. Einführung in Scheme

Jetzt geht es endlich los. In diesem Kapitel erlernen wir die Grundlagen der Programmierung mit Scheme.

3.1. Geklammerte Präfixnotation

Beim Betrachten von Scheme-Code fallen zuerst die vielen runden Klammern auf. Jeder Ausdruck wird grundsätzlich geklammert. Ein Zeichen zum Terminieren eines Ausdrucks entfällt somit. Für Funktionen und Operatoren wird stets die *Präfixnotation* verwendet. Die Möglichkeit, binäre Operatoren oder Funktionen in Infixnotation zu schreiben – wie es in Haskell möglich ist – besteht nicht. Zeilen- und Zeilenendkommentare werden durch ein Semikolon eingeleitet:

```
> (+ 1 2 3) ; Addition
6
> (+ 1 (/ 4 2) 3) ; geschachtelter Ausdruck
6
```

Inline- und Blockkommentare werden mit `#|` und `|#` umschlossen:

```
> (+ 1 #| nicht 2 |# 3)
4
> (+ 1 #| (/ 4 2) |# 3)
4
```

Innenliegende Ausdrücke oder Werte können zudem durch ein vorangestelltes `#;` auskommentiert werden:

```
> (+ 1 #; 2 3)
4
> (+ 1 #; (/ 1 2) 3)
4
```

Seit der Einführung des R⁷RS-Standards stehen außerdem eckige Klammern zur Verfügung. Diese können dazu genutzt werden, um die Lesbarkeit von Ausdrücken zu verbessern. Eine häufig anzutreffende Konvention ist, eckige Klammern nur bei bestimmten Ausdrücken wie `cond` einzusetzen. Runde und eckige Klammern sind dabei vollkommen gleichwertig. Eine öffnende Klammer muss jedoch mit derselben Klammerart wieder geschlossen werden.

3.2. Arithmetische Ausdrücke

Scheme kennt eine Vielzahl verschiedener Darstellungen für Zahlen. Es steht außerdem eine Unmenge von Rechenoperationen zur Verfügung, die wir hier aber nicht in ihrer Vollständigkeit behandeln werden:

```
> (+ 3 -2 0.7)
1.7
> (expt 3 2) ; Exponentiation
9
> (sqrt 9)
3
> (modulo 10 4)
2
```

Es stehen die üblichen arithmetischen Operatoren +, -, *, und / zur Verfügung. Reelle Zahlen können zudem in wissenschaftlicher Notation angegeben werden:

```
> (* 567 1e-2)
5.67
```

Auch mit komplexen Zahlen können wir rechnen:

```
> (sqrt -1)
0+1i
> (expt -i 2)
-1
> (+ 3+2i 5+5i)
8+7i
```

Rationale Zahlen können wir in der Form a/b angeben. Dabei ist / kein Operator, sondern Teil der Notation:

```
> (- 2 4/3)
2/3
```

Scheme kennt und erkennt bei Berechnungen sowohl *beliebige Genauigkeit* als auch die ungenaue Darstellung von Ergebnissen. Mit **exact?** können wir herausfinden, ob ein Ergebnis exakt ist oder nicht:

```
> (/ 1 3)
1/3
> (exact? (/ 1 3))
#t
> (/ 1 2.9)
0.3448275862068966
> (exact? (/ 1 2.9))
#f
```

Wahrheitswerte werden in Scheme durch `#t` und `#f` repräsentiert. Dabei steht `#f` für einen als „falsch“ ausgewerteten Ausdruck. Alle anderen Ausdrücke gelten als „wahr“:

```
> (not #t)
#f
> (not #f)
#t
> (not "false")
#f
```

Zum Vergleichen von Zahlen können wir die bekannten Operatoren nutzen:

```
> (= 7 3)
#f
> (> 7 3)
#t
> (<= 7 3)
#f
> (not (= 7 3))
#t
```

3.3. Zeichenketten

Zeichenketten (Strings) werden mit doppelten Anführungszeichen umschlossen. Einzelzeichen (Characters) können wir durch `#\c` gefolgt von einem Zeichen angeben:

```
> "Mueller-Luedenscheidt"
"Mueller-Luedenscheidt"
> #\c ; Character 'c'
#\c
```

Ein Character wie `#\c` ist dabei *kein* String. Umgekehrt ist der String `"s"` *kein* Character:

```
> (string? "s")
#t
> (string? #\c)
#f
> (char? #\c)
#t
> (char? "s")
#f
```

Es stehen viele Funktionen zum Arbeiten mit Zeichenketten bereit. So können zwei Strings mit `(string=? "abc" "def")` auf Gleichheit überprüft werden. Mit `string-append` werden Strings konkateniert:

```
> (string-append "Do" "lorem ipsum")
"Dolorem ipsum"
```

3.4. Funktionen und λ -Kalkül

Mit **define** kann ein Ausdruck an einen Bezeichner gebunden werden. Im einfachsten Fall hat eine solche Anweisung die Form $\langle \text{define } \textit{Bezeichner} \textit{ Ausdruck} \rangle$. Damit sind bereits Definitionen von Variablen und parameterlosen Funktionen möglich:

```
> (define wert 7)
> (define antwort (* 6 wert))
> antwort
42
```

Die allgemeine Syntax lautet:

$$\langle \text{define } \langle \textit{Bezeichner} \textit{ Arg}_1 \dots \textit{ Arg}_n \rangle \textit{ Ausdruck}_1 \dots \textit{ Ausdruck}_n \rangle$$

Eine Funktion zum Addieren zweier Zahlen könnten wir so implementieren:

```
> (define (addiere x y) (+ x y))
    (addiere 3 4)
7
```

Die Ausdrücke einer Definition werden der Reihe nach ausgewertet. Erst das Ergebnis des letzten Ausdrucks bildet dann den Rückgabewert der Funktion:

```
> (define (addiereUndQuadriere x y)
    (define summe (+ x y))
    (expt summe 2))
    (addiereUndQuadriere 3 4)
49
```

Natürlich implementiert Scheme auch den λ -Kalkül (Lambda-Kalkül). Die Syntax ähnelt der von Funktionsdefinitionen, wobei der Bezeichner entfällt:

$$\langle \text{lambda } \langle \textit{Arg}_1 \dots \textit{ Arg}_n \rangle \textit{ Ausdruck}_1 \dots \textit{ Ausdruck}_n \rangle$$

Das Quadrieren einer Zahl können wir dann als anonyme λ -Funktion umsetzen:

```
> ((lambda (x) (expt x 2)) 7)
49
```

Unsere Funktion `addiereUndQuadriere` könnten wir dann beispielsweise umschreiben zu:

```
(define (addiereUndQuadriere x y)
  (define summe (+ x y))
  ((lambda (x) (expt x 2)) summe))
```


3.5. Kontrollstrukturen

Verzweigungen können mit **if** oder **cond** realisiert werden. Ersteres funktioniert denkbar einfach:

 $\langle \text{if Bedingung Aktion Sonst} \rangle$

Trifft die Bedingung zu, wird die Aktion ausgeführt, ansonsten der darauffolgende Ausdruck. Damit können wir beispielsweise eine Betragsfunktion schreiben:

```
> (define (betrag x) (if (>= x 0) x (* -1 x)))
> (betrag 3)
3
> (betrag -1)
1
```

Eine else-if-Anweisung, wie wir sie aus anderen Sprachen kennen, steht in Scheme nicht zur Verfügung. Um eine if-elseif-Konstruktion zu erreichen, könnte man **if**-Anweisungen mehrfach schachteln. Weil das schnell unübersichtlich wird, gibt es die **cond**-Anweisung:

 $\langle \text{cond Klausel}_1 \dots \text{Klausel}_n \rangle$

Eine Klausel hat immer die Form $\langle \text{Bedingung Ausdruck}_1 \dots \text{Ausdruck}_n \rangle$. Die letzte Klausel kann alternativ eine **else**-Anweisung statt einer Bedingung enthalten: $\langle \text{else Ausdruck}_1 \dots \text{Ausdruck}_n \rangle$. Nun definieren wir eine Funktion, die zwei Werte vergleicht und eine entsprechende Ausgabe erzeugt:

```
> (define (vergleiche x y)
  (cond [(> x y) "groesser"]
        [(< x y) "kleiner"]
        [else "gleich"]))
> (vergleiche 7 3)
"groesser"
```

Es ist üblich, die Klauseln in eckige statt runde Klammern einzufassen. Das verbessert die Lesbarkeit.

3.6. Listen und Paare

Der einfachste Weg, eine *Liste* zu erzeugen, ist, die Listenelemente zwischen runden Klammern aufzuzählen. Der Versuch, durch $(abc\ d\ e\ f)$ eine Liste zu definieren, führt jedoch zu einer Fehlermeldung des Interpreters. Das semantische Problem liegt darin, dass nicht klar ist, ob es sich bei abc um einen Funktionsaufruf oder um das erste Listenelement

handeln soll. Abhilfe verschafft das **quote**-Schlüsselwort. Üblicherweise wird jedoch `'()` als alternative Schreibweise verwendet. Beide Varianten sind gleichwertig:

```
> (quote (abc d e f))
(abc d e f)
> '(abc d e f)
(abc d e f)
```

Listen lassen sich aber auch mit **list** erzeugen. Dabei handelt es sich – im Gegensatz zu **quote** – um eine Funktion. Daher werden die Argumente *vorher* ausgewertet. Bei **quote** ist das *nicht* der Fall, denn **quote** ist keine Funktion, sondern ein Sprachelement von Scheme:

```
> (list 1 (+ 1 1) (+ 1 2)) ; Ausdrücke werden ausgewertet
(1 2 3)
> '(1 (+ 1 1) (+ 1 2))      ; Ausdrücke werden NICHT ausgewertet
(1 (+ 1 1) (+ 1 2))
```

Die *leere Liste* wird durch `()` repräsentiert und mit `'()` in einem Ausdruck verwendet. Mit **car** wird das erste Listenelement zurückgegeben und mit **cdr** der Rest der Liste:

```
> (car '(1 2 3 4 5))
1
> (cdr '(1 2 3 4 5))
(2 3 4 5)
> (cdr '(1)) ; der Rest dieser Liste ist die leere Liste
()
```

Außerdem gibt es noch *Paare*, welche mittels **cons** erzeugt werden. Dieses verlangt genau zwei Argumente:

```
> (define paar (cons "Ein" "Paar"))
> paar
("Ein" . "Paar")
> (car paar)
"Ein"
> (cdr paar)
"Paar"
```

Zwischen Listen und Paaren besteht ein wichtiger Zusammenhang: Ein Paar verbindet zwei beliebige Werte. Eine Liste besteht wiederum aus geschachtelten Paaren:

```
> (list 1 2)
(1 2)
> (cons 1 (cons 2 '())) ; dieselbe Liste aus Paaren konstruiert
(1 2)
```

Genauer ausgedrückt sind Listen Paare, dessen zweites Element wiederum eine Liste ist. Listen sind also verkettete Paare. Das letzte Element ist dabei immer die leere Liste:

```
> (pair? (list 1 2))
#t
> (pair? (cons 1 (cons 2 '())))
#t
```

Paare sind im Umkehrschluss jedoch nicht immer Listen:

```
> (list? (cons 1 2))
#f
```

In den nächsten beiden Abschnitten lernen wir verschiedene Möglichkeiten kennen, mit Listen zu arbeiten.

3.7. Mapping

Beim *Mapping* wird eine Funktion auf die Elemente einer oder mehrerer Listen angewendet. Die Syntax lautet:

$\langle \text{map Funktion Liste}_1 \dots \text{Liste}_n \rangle$

Betrachten wir zunächst den einfachsten Fall mit nur einer Liste. Hier möchten wir jedes Listenelement quadrieren:

```
> (define (quadrat x) (* x x))
      (map quadrat '(1 2 3 4 5))
(1 4 9 16 25)
```

Beim Mapping mit mehreren Listen müssen diese die gleiche Anzahl an Elementen aufweisen, und die Funktion muss genauso viele Argumente haben, wie es Listen gibt. Dabei gehen die Elemente der ersten Liste der Reihe nach immer als erstes Argument in die Funktion ein, die Elemente der zweiten Liste als zweites Argument, und so weiter. Das Ergebnis ist eine einzige Liste. Im folgenden Beispiel addieren wir paarweise die Elemente zweier Listen. Dazu verwenden wir eine λ -Funktion:

```
> (map (lambda (x y) (+ x y))
      '(1 2 3 4)
      '(5 6 7 8))
(6 8 10 12)
```

Die neu entstandene Liste enthält die Ergebnisse der paarweisen Addition der Elemente beider Listen.

3.8. Folding

Beim *Folding* wird eine Funktion mit einem Startwert auf die Elemente einer Liste angewendet. Anders als beim Mapping ist das Ergebnis aber keine Liste, sondern ein Wert. Man unterscheidet zwischen *rechtem* und *linkem Folding*. Als Beispiel sei die Berechnung der Summe der Elemente der Liste '(1 2 3 4 5)' angeführt. Mathematisch betrachtet würden die Elemente beim linken Folding wie folgt geklammert:

$$(((1 + 2) + 3) + 4) + 5$$

Beim rechten Folding würde von rechts geklammert:

$$1 + (2 + (3 + (4 + 5)))$$

Die Funktion – hier die Addition – wird dann wiederholt auf die innersten beiden Elemente angewendet, bis die Liste komplett abgearbeitet ist. Damit ist auch klar, dass die Funktion über zwei Parameter verfügen muss. Bei der Addition ist das Ergebnis in beiden Fällen dasselbe. Bei Operationen, bei denen die Auswertreihenfolge eine Rolle spielt, können linkes und rechtes Folding dagegen zu unterschiedlichen Ergebnissen führen. Der Einsatz von Folding ist nicht auf mathematische Operationen beschränkt!

Wir erinnern uns, dass eine Liste letztendlich aus geschachtelten Paaren besteht. Grafisch lässt sich das Folding daher als Baum darstellen (Abbildung 1). Nun wird klar, warum beim Folding ein Startwert angegeben werden muss: Das innerste Element ist eine leere Liste, die Funktion erwartet allerdings zwei passende Argumente.

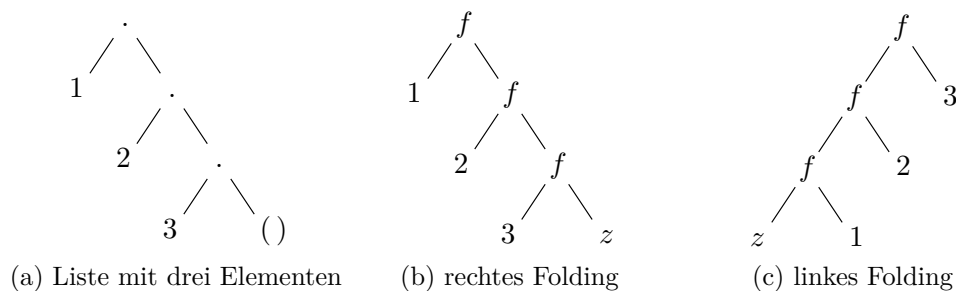


Abbildung 1: Anwendung einer Funktion f mit dem Startwert z auf eine Liste (a) durch rechtes (b) und linkes (c) Folding.

Die Syntax beim Folding ähnelt der beim Mapping. Zusätzlich wird der Startwert angegeben. Es folgt die Syntax für linkes Folding (beim rechten Folding wird entsprechend **fold-right** verwendet):

fold-left Funktion Startwert Liste₁ ... Liste_n

Mit der folgenden Anweisung bilden wir die Listensumme. Als Startwert geben wir 0 als das neutrale Element der Addition an. Zuerst müssen wir die `rnrs`-Bibliothek laden:

```
> (require rnrs)
> (fold-left (lambda (x y) (+ x y)) 0 '(1 2 3 4))
10
```

Beim Folding *einer* Liste muss die Funktion zwei Argumente haben. Für jede *weitere* Liste muss die Funktion genau wie beim Mapping ein zusätzliches Argument aufnehmen können.

Damit endet unsere Einführung in Scheme. Im nächsten Kapitel lernen wir die Racket-Grafikbibliothek kennen.

4. Bilderzeugung

Wie versprochen schauen wir uns jetzt eine Grafikbibliothek an, mit welcher wir geometrische Objekte erzeugen können. Dabei werden wir zwar keine neuen Sprachelemente kennenlernen, betrachten aber ein Anwendungsgebiet, für welches sich besonders funktionale Programmiersprachen sehr gut eignen. Schließlich lassen sich geometrische Figuren durch Funktionen beschreiben.

Bei der verwendeten Bibliothek handelt es sich eigentlich um eine Racket-Bibliothek. Auch wenn Racket und Scheme miteinander nicht direkt kompatibel sind, können wir diese Bibliothek trotzdem nutzen, da sie mehrere Lisp-Dialekte unterstützt. Dem Racket-Interpreter muss lediglich mitgeteilt werden, dass er im Scheme-Modus arbeiten soll. Das haben wir bereits mittels `#lang scheme` erledigt [FF, *The Racket Guide: 23. Dialects of Racket and Scheme*].

Zunächst sehen wir, wie man mathematische Funktionsgraphen plotten kann. Anschließend nutzen wir Rekursion, um geometrische Gebilde zu erzeugen.

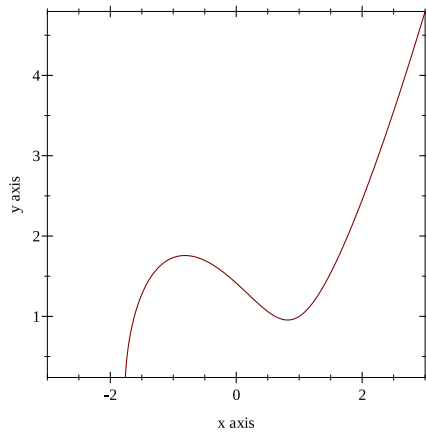
4.1. Funktionsgraphen

Als Erstes wollen wir eine elliptische Kurve plotten. Diese Kurven kommen beispielsweise in der Kryptographie bei asymmetrischen Verschlüsselungsverfahren zum Einsatz. Sie haben die Form $y^2 = x^3 + ax + b$ mit $x, y \in \mathbb{R}$ und sind leicht an ihrem symmetrischen Aussehen zu erkennen [SPS11, S. 136].

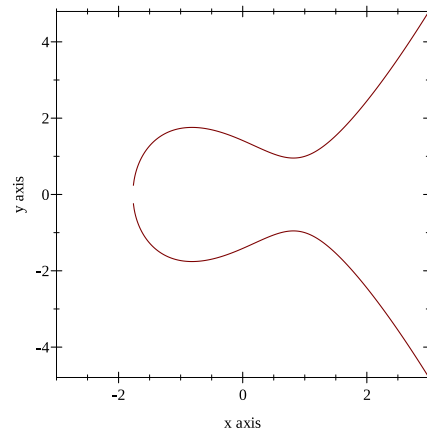
Eine naive Implementierung der Kurve $y^2 = x^3 - 2x + 2$ könnte etwa so aussehen:

```
1 (require plot)
2
3 ;; Elliptische Kurve:  $y^2 = x^3 - 2x + 2$ 
4 (define (elliptic x) (sqrt (+ (expt x 3) (* -2 x) 2)))
5
6 (plot (function elliptic -3 3)) ; Funktionsgraphen plotten
7 (plot-file (function elliptic -3 3) "kurve.svg") ; SVG-Datei erzeugen
```

Zuerst wird das `plot`-Modul geladen. Zur Berechnung der Funktionswerte der elliptischen Kurve wird die Funktion `elliptic` definiert. Mit `plot` wird der Funktionsgraph dann direkt im Interpreterfenster dargestellt. Dabei wird `elliptic` als das erste Argument an `function` übergeben, gefolgt von zwei Werten, die den Bereich der x -Achse festlegen, der dargestellt werden soll. Von `function` wird ein *Renderer*-Objekt zurückgegeben, das dann wiederum von `plot` verarbeitet wird. Es können auch noch weitere Optionen zum Formatieren der Ausgabe angegeben werden. Mit `plot-file` kann zudem eine Vektorgrafik erzeugt werden [FF, *Plot: Graph Plotting*]. Das Ergebnis zeigt Abbildung 2a.



(a) Die untere Hälfte fehlt.



(b) Vollständige Darstellung

Abbildung 2: Die elliptische Kurve $y^2 = x^3 - 2x + 2$ ohne untere Hälfte (a) aufgrund einer fehlerhaften Implementierung und vollständig (b) nach Beheben des Programmierfehlers.

Offensichtlich ist unsere Implementierung fehlerhaft. Das Ergebnis erinnert nicht an eine elliptische Kurve. Wir haben nicht bedacht, dass beim Umformen der Gleichung gilt:

$$y^2 = x^3 - 2x + 2$$

$$\Leftrightarrow y_{1,2} = \pm \sqrt{x^3 - 2x + 2}$$

Somit bildet die Funktion jeden x -Wert auf *zwei* Funktionswerte ab: einen positiven und einen negativen y -Wert.² Das können wir erreichen, indem wir den Graphen spiegeln. Hierbei genügt die einmalige Angabe des Wertebereichs:

```
(plot (list (function elliptic -3 3)
            (function (lambda (x) (- (elliptic x))))))
```

An `plot` kann eine Liste von Funktionen übergeben werden, welche dann alle im selben Koordinatensystem dargestellt werden. Hier geben wir eine λ -Funktion an, die die Funktionswerte von `elliptic` mit negativem Vorzeichen zurückgibt. Damit ist unsere elliptische Kurve vollständig, wie in Abbildung 2b zu sehen ist.³

² Daher handelt es sich streng genommen um keine Funktion, sondern um eine Relation.

³ Die Kurve hat genau eine Nullstelle. Die Lücke der Kurve beim x -Achsendurchgang ist vermutlich auf einen Darstellungsfehler oder auf Ungenauigkeiten bei der Berechnung der Kurvenpunkte zurückzuführen.

4.2. Geometrische Objekte

Als Nächstes werden wir eine geometrische Figur durch Rekursion erzeugen. Das `image`-Modul stellt Funktionen bereit, um einfache Formen wie Kreise oder Rechtecke zu zeichnen und anzuordnen. Die Figur soll so aussehen, wie in Abbildung 3 dargestellt.

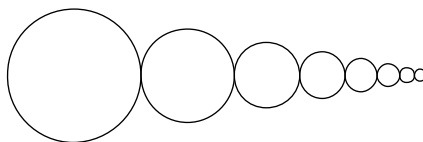


Abbildung 3: Eine rekursiv erzeugte Figur. Die Rekursionstiefe bestimmt die Anzahl der Kreise.

In der imperativen Programmierung würde man die Kreise eventuell mithilfe einer Schleife erzeugen. Hier verwenden wir Rekursion:

```
1 (require 2htdp/image)
2
3 ;; Kreis mit Radius r
4 (define (kreis r) (circle r "outline" "black"))
5
6 ;; Rekursionstiefe n, Startradius r
7 (define (kreise n r)
8   (cond ((= n 0) (kreis r))
9         (else (beside (kreis r)
10                        (kreise (- n 1) (* r 0.7))))))
11
12 (freeze (kreise 7 50)) ; Ausgabe im Interpreter
13 (save-svg-image (kreise 7 50) "kreise.svg") ; Vektorgrafik
```

Zuerst definieren wir die Funktion `kreis`, um einen einzelnen Kreis zu zeichnen. Das passiert mit der `circle`-Funktion, welcher der Radius und Hinweise zur Darstellung übergeben werden. Die Funktion `kreise` erhält neben dem Startradius einen Anfangswert für den Rekursionstiefenzähler. Auf diese Weise lässt sich die Anzahl der zu zeichnenden Kreise variabel anpassen. Mit `beside` können mehrere Grafikobjekte nebeneinander positioniert werden. Die Reihenfolge der Argumente bestimmt die Reihenfolge der Anordnung. Es wird immer ein einzelner Kreis erstellt (Zeile 9), gefolgt vom rekursiven Funktionsaufruf. Dabei wird der Zähler dekrementiert und der Radius des nächsten Kreises um den Faktor 0.7 skaliert. Im allerletzten rekursiven Aufruf ($= n 0$) wird der letzte Kreis gezeichnet. Mit `freeze` wird das gesamte Gebilde als Bitmap im Interpreter ausgegeben oder alternativ mit `save-svg-image` in eine Vektorgrafik überführt.

Spätestens im nächsten Beispiel wird klar, warum man in vielen Fällen ohne Rekursion nicht ans Ziel kommt.

4.3. Fraktale

Die nächste Figur wird etwas komplizierter. Das in Abbildung 4 gezeigte Fraktal ist ein *exakt selbstähnliches* Gebilde. Das bedeutet, dass beim Hineinzoomen in das Bild immer wieder dieselbe Struktur zu erkennen ist. Fraktale werden meist durch rekursive Verfahren generiert [Wik20].

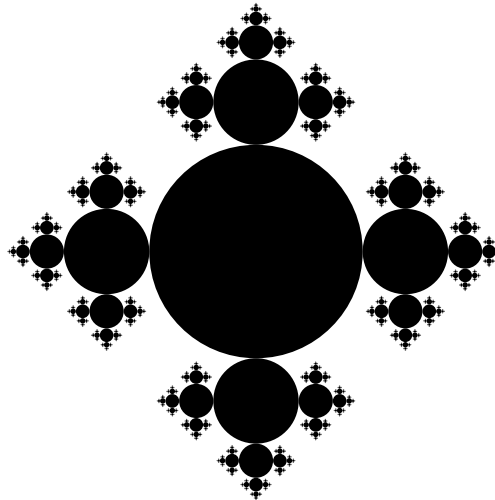


Abbildung 4: Ein Fraktal mit exakter Selbstähnlichkeit. Die Struktur bleibt beim Vergrößern erhalten.

Glücklicherweise haben wir bereits gute Vorarbeit geleistet. Die nachfolgende Implementierung ist weitestgehend mit dem Programm aus Abschnitt 4.2 identisch. Lediglich der **else**-Teil ab Zeile 7 unterscheidet sich:

```
1  (require 2htdp/image)
2
3  (define (kreis r) (circle r "solid" "black"))
4
5  (define (fraktal n r)
6    (cond ((= n 0) (kreis r))
7          (else (define fractus (fraktal (- n 1) (* r 0.4)))
8                  (above fractus                                ; oben
9                          (beside (rotate 90 fractus)           ; links
10                                   (kreis r)                     ; Zentrum
11                                   (rotate -90 fractus)))))) ; rechts
12
13  (freeze (fraktal 7 100))
```

Vom großen Kreis in der Mitte abgesehen, hat jeder weitere Kreis genau drei kleinere „Satelliten“. Dieses Muster wiederholt sich dann. Um diese Satelliten zu erzeugen, wird also

letztendlich die Funktion `fraktal` rekursiv aufgerufen. Um Redundanz zu vermeiden und den Code übersichtlicher zu gestalten, wurde dafür eine Hilfsfunktion `fractus` definiert. Mit `above` und `beside` werden die Satelliten um den gerade aktuellen zentralen Kreis (Zeile 10) herum angeordnet. Dabei müssen zwei von ihnen mittels `rotate` um 90° gedreht werden.

Was noch fehlt, ist der Ableger unterhalb des zentralen Hauptkreises, denn dieser bietet Platz für vier Satelliten. Schlimm ist das zwar nicht, kann aber durch folgende Anweisung erledigt werden:

```
(freeze (above (fraktal 7 100) (rotate 180 (fraktal 6 40))))
```

Dazu wird einfach ein zweites kleineres Fraktal um 180° gedreht und unter die Hauptfigur gesetzt. Dabei müssen Rekursionstiefe und Radius entsprechend angepasst werden. Der Faktor, um den die Satelliten verkleinert werden (0.4), ist übrigens bewusst gewählt. Wählt man diesen zu groß (bspw. 0.5), liegen die Kreise nicht mehr direkt aneinander. Diese Lücken entstehen, weil ein Ableger (mit all seinen eigenen Ablegern) dann breiter werden kann als der Kreis, an dem er hängt. Das Problem lässt sich zwar lösen, aber nicht mit den hier vorgestellten primitiven Funktionen `above` und `beside`.

Damit endet auch der praktische Teil unserer Einführung in Scheme.

5. Zusammenfassung

Nun haben wir die Grundlagen von Scheme erlernt und sind in der Lage, einfache Programme selbst zu schreiben. Wir haben uns an die vielen runden Klammern gewöhnt und einen Eindruck von anderen (wesentlicheren) Unterschieden zwischen Scheme und Haskell gewonnen. Außerdem haben wir die Racket-Grafikbibliothek kennengelernt. Dabei haben wir festgestellt, wie leicht es ist, mathematische und geometrische Sachverhalte mit einer funktionalen Programmiersprache abzubilden.

Ausblick

Nach dem Durcharbeiten dieser Einführung sollte es für uns kein Problem sein, uns auch die fortgeschritteneren Sprachelemente und Konzepte von Scheme anzueignen. Hier sei noch einmal auf die in Abschnitt 2.2 empfohlene Literatur verwiesen. Es ist immer vorteilhaft, mehr als nur einen Vertreter eines Programmierparadigmas zu kennen. Das versetzt uns in die Lage, bei möglichen Projekten fundierte Entscheidungen treffen zu können, wie beispielsweise die Wahl der Programmiersprache. Detailkenntnisse in jeder Sprache sind dafür nicht erforderlich. Erste Erfahrungen, wie wir sie hier gesammelt haben, sind dafür häufig schon ausreichend.

Literaturverzeichnis

- [ASS96] Harold Abelson, Gerald Jay Sussman und Julie Sussman. *Structure and Interpretation of Computer Programs*. 2. Aufl. The MIT Press, 1996. URL: <https://mitpress.mit.edu/sites/default/files/sicp/index.html>.
- [Dyb09] R. Kent Dybvig. *The Scheme Programming Language*. 4. Aufl. The MIT Press, 2009. URL: <https://www.scheme.com/tspl4/>.
- [FF] Matthew Flatt und Robert Bruce Findler. *Racket Documentation*. PLT Inc. URL: <https://docs.racket-lang.org/> (besucht am 29.06.2020).
- [FF95] Daniel P. Friedman und Matthias Felleisen. *The Little Schemer*. 4. Aufl. The MIT Press, 1995.
- [SCG13] Alex Shinn, John Cowan und Arthur A. Gleckler. *Scheme Reports Process*. 2013. URL: <http://www.scheme-reports.org/> (besucht am 07.07.2020).
- [SPS11] Stephan Spitz, Michael Pramateftakis und Joachim Swoboda. *Kryptographie und IT-Sicherheit. Grundlagen und Anwendungen*. 2. Aufl. Wiesbaden: Springer, 2011.
- [Tio20] Tiobe. *TIOBE programming community index*. TIOBE Software BV. Juni 2020. URL: <https://www.tiobe.com/tiobe-index/> (besucht am 24.06.2020).
- [Wik20] Wikipedia. *Fractal*. 2. Juli 2020. URL: <https://en.wikipedia.org/w/index.php?title=Fractal&oldid=965564663> (besucht am 04.07.2020).