

KAPITEL 1

t.Zero: Deklarative Programmierung

Deklarativ: Eigenschaft von Sprechakten (= deklarativen Akten/Deklarationen), die, unter geeigneten Bedingungen verwendet, eine neue Wirklichkeit herstellen. (<http://de.wiktionary.org/wiki/deklarativ>)

Dieses Kapitel behandelt t.Zero, eine extrem einfache Miniatur-Programmiersprache. Der Name ist, anders als beispielsweise t.Scheme oder t.Java, nicht von einer „richtigen“ Programmiersprache abgeleitet, sondern steht für t_0 , den Startpunkt der t.Sprachfamilie, die im Folgenden entwickelt wird.

t.Zero ist eine Art Nulldiät für Programmierer: Die (fast) einzigen Elemente der Sprache sind Zahlen und Funktionen. Es gibt keine Variablen, keine Wertzuweisungen, keine While- oder For-Schleifen, keine Deklarationen neuer Typen oder Klassen, keines der zahllosen „Features“, die man von anderen Programmiersprachen her kennt.

Trotzdem ist t.Zero eine im Sinne der Informatik vollständige Programmiersprache: Man kann damit alles berechnen, was auf irgendeinem Computer berechnet werden kann. Und gerade weil die Sprache so schlicht ist, kann man an ihr lernen, wie mit minimalen Mitteln interessante Dinge zuwege gebracht werden können.

Im Folgenden wird vorausgesetzt, dass die Software zu diesem Buch bereits installiert ist (vgl. Anhang A).

1.1 Sprachelemente

1.1.1 Zahlen

Alle in diesem Buch definierten Sprachen sind interpretiert: Programme werden nicht übersetzt, sondern von einem Interpreter unmittelbar ausgeführt. Man kann mit solchen Sprachen interaktiv arbeiten.

Der *t.Zero*-Interpreter wird mit dem Kommando *tzero* aufgerufen. Als erstes Lebenszeichen gibt er eine Begrüßung und eine Eingabeaufforderung aus (einen *Prompt* in Form eines Pfeils *->*), dann wartet er auf Eingaben:

```
$ tzero
*** tZero ***
To quit, type '(quit)'
->
```

Hier wurde der Befehl *tzero* in einen Kommandozeileninterpreter (eine *Shell*) eingegeben. Das Dollarzeichen ist die Eingabeaufforderung dieser *Shell*.

Jede Eingabe für *t.Zero* wird mit dem Drücken der Return-Taste abgeschlossen. Auf legale Eingaben reagiert der Interpreter mit der Ausgabe eines Resultats.

Die einzigen Datentypen von *t.Zero* sind Zahlen und Wahrheitswerte. Man kann auf Zahlen die Grundrechenarten anwenden, das Symbol für die Rechenoperation muss dabei vorangestellt werden. Außerdem muss man – auf den ersten Blick sehr umständlich – Operator und Operanden in runde Klammern setzen:

```
-> 37
37
-> (- 37 27)
10
-> (* 37 27)
999
-> (/ 111 37)
3
-> (+ (* 3 3) (* 4 4))
25
```

Zahlen werden in Dezimaldarstellung eingegeben, mit beliebig vielen Stellen vor und nach dem Dezimalpunkt (ein Dezimalkomma ist nicht erlaubt):

```
-> -0000.00000000000000000005
-0.00000000000000000005
-> 1.
1
```

Die Resultate von Rechnungen sind in *t.Zero* immer exakt. Wenn ein Ergebnis sich nicht als Dezimalzahl darstellen lässt, wird ein gekürzter Bruch zurückgegeben:

```
-> (/ 1 3)
1/3
-> (/ 11 187)
1/17
```

Man kann Bruchzahlen auch direkt eingeben. Dabei wird nicht gekürzt:

```

-> 2/34
2/34
-> (+ 2/3 3/4)
17/12
-> 2 / 17
Error: Trailing input

```

Bei der letzten Eingabe liest t.Zero eine 2 und versteht diese wegen des nachfolgenden Leerzeichens als komplette Eingabe. Da in jeder Zeile nur eine einzige Eingabe stehen darf, beschwert sich der Interpreter über die weiteren Zeichen. Die Eingabe 2/34 enthält keine Leerzeichen, sie stellt eine einzelne Zahl dar.

Konzeptionell betrachtet ist in t.Zero jede Zahl ein Bruch mit einem Zähler und einem Nenner. Mit den Operatoren `num` und `den` (für Numerator und Denominator) kann man Zähler und Nenner abfragen:

```

-> (num 0.5)
1
-> (den 0.5)
2

```

Will man Brüche in Dezimaldarstellung bringen, so muss man runden. Dazu dient der Operator `round`. Er erwartet zwei Argumente, die zu rundende Zahl und die gewünschte Anzahl Nachkommastellen:

```

-> (round 1/19 20)
0.05263157894736842105

```

Das vorangegangene Resultat kann man jeweils unter dem Namen `%` weiterverwenden:

```

-> (* 19 %)
0.9999999999999999995
-> (- 1 %)
0.0000000000000000005

```

Die jeweils letzte Eingabe hat den Namen `@`. Man braucht sie viel seltener als `%`, eigentlich nur dann, wenn man nochmal sehen will, was man eigentlich eingegeben hatte.

Gerundet wird übrigens immer zur Null hin:

```

-> (round -7.53 1)
-7.5
-> (round 7.53 3)
7.530

```

1.1.2 Funktionen

Bisher hat t.Zero zu jeder Eingabe unter Verwendung der Operatoren `+`, `-`, `*`, `/` und `round` unmittelbar einen Wert berechnet.

Man kann sich sehr einfach weitere eigene Funktionen verschaffen. Eine Funktion `trunc` zum Abschneiden der Nachkommastellen einer Zahl definiert man zum Beispiel so:

```

-> (define (trunc x) (round x 0))
function[trunc]
-> (trunc -3.7)
-3

```

Funktionsdefinitionen dürfen sich über mehr als eine Zeile erstrecken:

```
-> (define (square n)
      (* n n))
function[square]
-> (square 111111111)
12345678987654321
```

Funktionen können natürlich auch mehrere Argumente haben:

```
-> (define (square-sum x y)
      (+ (square x) (square y)))
function[square-sum]
-> (square-sum 3 4)
25
```

Auch Funktionen ohne Argumente sind erlaubt. Man kann sie zur Speicherung von Konstanten verwenden, was manchmal praktisch ist, weil es in *t.Zero* keine Variablen gibt:

```
-> (define (Pi) 3.1415926535)
function[Pi]
-> (Pi)
3.1415926535
```

Die allgemeine Form einer Funktionsdefinition ist

(define <Funktionskopf> <Funktionsrumpf>).

Der Funktionskopf besteht aus einer geklammerten Liste von Namen. Der erste Name ist der Funktionsname, die übrigen sind die *formalen Parameter* der Funktion. Beim Aufruf der Funktion werden sie durch die *aktuellen Parameter* ersetzt und der Funktionsrumpf wird mit diesen Parameterwerten ausgewertet. Eine Funktionsdefinition wird gespeichert, der Interpreter erinnert sich bei den weiteren Auswertungen daran.

Die Namen von Funktionen dürfen Sonderzeichen enthalten:

```
-> (define (++ x) (+ 1 x))
function[++]
-> (++ 0)
1
```

Die ungewohnte Art, bei einer Rechenoperation den Operator voranzustellen, wirkt jetzt vielleicht nicht mehr ganz so seltsam. Eine Rechenoperation wird genauso behandelt wie der Aufruf einer selbst definierten Funktion: (+ 3 7) ist ein Aufruf der Funktion + mit den Argumenten 3 und 7.

Am Anfang ist es möglicherweise irritierend, dass man (*f x*) schreiben muss, nicht *f(x)* wie in den meisten anderen Programmiersprachen und in der Mathematik. Diese *Präfixnotation* geht auf die Sprache Lisp zurück. Ihre Vorteile werden sich in Kapitel 2 zeigen.

Alle gültigen Eingaben von *t.Zero*, seien es Zahlen oder unter Verwendung von Klammern geschriebene zusammengesetzte Eingaben, werden als *Ausdrücke* bezeichnet. Der Interpreter wertet Ausdrücke aus.

Auch eine Funktionsdefinition ist ein Ausdruck, deshalb wird als Wert eine Funktion zurückgegeben. Trotzdem haben Funktionsdefinitionen eine Sonderstellung unter den *t.Zero*-Ausdrücken. Man gibt sie nicht ein, um ihren Wert zu berechnen, sondern damit der Interpreter sie sich merkt.

Der Wert könnte ebenso gut fehlen, er ist nur eine Bestätigung dafür, dass die Definition gespeichert wurde.

1.1.3 Bedingte Ausdrücke

Man kann Zahlen mit = und < vergleichen. Das Resultat hat den Datentyp Boolean und kann in einem bedingten Ausdruck verwendet werden:

```
-> (= 1 1.0)
true
-> (< 1 (/ 1 1.00000001))
false
-> (if (< 0.5 1/2) 3 4)
4
```

Eine Eingabe der Form (if <condition> <expr> <alt-expr>) ist ein *bedingter* Ausdruck. Ihr Wert ist entweder der Wert von <expr> oder der von <alt-expr>, je nachdem, ob <condition> den Wert true oder false hat. Deshalb muss für <condition> immer ein Ausdruck stehen, der einen Wert des Typs Boolean hat.

Ein bedingter Ausdruck (if ...) steht, anders als eine if-Anweisung in Java, für einen Wert. In Java hat der ?-Ausdruck <condition> ? <expr> : <alt-expr> die entsprechende Funktion.

Die Abfrage (= 1 1.0) zeigt, dass der Vergleichsoperator = seine Argumente im mathematischen Sinn vergleicht. Es macht keinen Unterschied, ob es sich um Zahlen mit oder ohne Dezimalpunkt oder um Brüche handelt.

Von dem Typ Boolean gibt es nur die beiden Werte true und false. Es sind keine Schlüsselwörter wie in Java. In t.Zero darf man weder (if true <expr> <alt-expr>) noch (if <condition> true <alt-expr>) schreiben. Der Interpreter würde versuchen, den Ausdruck true auszuwerten, und da es in t.Zero keine Variablen und keine Zahlkonstanten gibt, würde das scheitern.

Als Ersatz kann man sich, wenn man will, argumentlose Funktionen mit den Namen true und false definieren:

```
-> (define (true) (= 0 0))
function[true]
-> (define (false) (= 0 1))
function[false]
```

t.Zero kennt keine Operatoren für Abfragen auf Ungleichheit, ≤-Beziehung oder logische Operatoren. Man kann sie sich aber leicht beschaffen:

```
-> (define (not x) (if x (false) (true)))
function[not]
-> (define (and x y) (if x (if y (true) (false)) (false)))
function[and]
-> (define (or x y) (if x (true) (if y (true) (false))))
function[or]
-> (define (# x y) (not (= x y)))
function[#]
-> (define (<= x y) (or (< x y) (= x y)))
function[<=]
```

1.1.4 Rekursion

t.Zero-Funktionen dürfen rekursiv definiert sein. Das Standardbeispiel der Fakultätsfunktion $n! = 1 \cdot 2 \cdot \dots \cdot n$ sieht so aus:

```

-> (define (factorial n)
      (if (= n 0) 1
          (* n (factorial (- n 1)))))
function[factorial]
-> (factorial 5)
120
-> (factorial (+ 1 (* 6 6)))
13763753091226345046315979581580902400000000

```

Gerade bei rekursiven Funktionen ist es manchmal hilfreich, den Verlauf einer Auswertung zu beobachten. Zu diesem Zweck kann man Funktionen in den *Tracing-Modus* versetzen:

```

-> (trace factorial)
Tracing mode for function[factorial] is on
-> (factorial 3)
Call    (factorial 3)
Call    (factorial 2)
Call    (factorial 1)
Call    (factorial 0)
Return  1 from (factorial 0)
Return  1 from (factorial 1)
Return  2 from (factorial 2)
Return  6 from (factorial 3)
6

```

Zu Beginn jedes Aufrufs einer Funktion im Tracing-Modus werden die Argumentwerte angezeigt; wenn der Aufruf beendet ist, wird das Resultat ausgegeben. Im obigen Beispiel kann man verfolgen, wie die einzelnen Aufrufe von `factorial` in der umgekehrten Reihenfolge ihres Beginns beendet werden.

Der Operator `trace` ist ein Schalter. Bei einem zweiten Aufruf versetzt er sein Argument wieder in den Normalmodus:

```

-> (trace factorial)
Tracing mode for function[factorial] is off
-> (factorial 50)
30414093201713378043612608166064768844377641568960512000000000000

```

Der Tracing-Modus hat keinerlei Auswirkungen auf die Ergebnisse. Der Trace-Operator verändert die Ausdrucksmöglichkeiten der Programmiersprache nicht.

1.1.5 Fehler

Fehlerhafte Eingaben quittiert der Interpreter mit einer Fehlermeldung:

```

-> (+ 1 °)
Error: Illegal token
-> )
Error: Expression cannot begin with ')'
-> (factorial 3 7)
Error: function[factorial] expects 1 argument
-> (factorial -1)
Error: Stack overflow

```

Diese Fehler sind, auch wenn die Fehlermeldungen das nicht anzeigen, von sehr verschiedener Art:

- Die Eingabe `'(+ 1 °)'` ist ein Fehler auf der *lexikalischen* Ebene der Sprache. Anstelle einer Null wurde das Gradzeichen eingetippt, das nicht zu den in t.Zero erlaubten Sonderzeichen gehört.
- Die Eingabe `')` enthält kein unerlaubtes Zeichen, ist aber *syntaktisch* fehlerhaft: Klammern müssen in der Eingabe immer paarweise auftreten.
- Da im Kopf von `factorial` in der Funktionsdefinition genau ein formaler Parameter vorkam, weist der Interpreter den Aufruf `'(factorial 3 7)'` zurück. Die Eingabe ist zwar lexikalisch und syntaktisch korrekt, lässt sich aber trotzdem nicht auswerten. Syntaktisch korrekte, nicht auswertbare Eingaben sind *semantische* Fehler.
- Der Aufruf `'(factorial -1)'` führt zu einer nicht endenden Rekursion, die irgendwann den Systemstack überlaufen lässt. Auch das ist ein semantischer Fehler.

Die Installation einer guten Fehlerbehandlung zählt zu den wichtigsten Aufgaben bei der Implementierung einer Programmiersprache. Fehler sollten ein Programm nicht zum Absturz bringen, sie sollten rechtzeitig erkannt und sinnvoll kommentiert werden. Wir werden auf das Thema noch ausführlich zu sprechen kommen.

Ein spezielles Problem ist die Frage, wie der Interpreter auf unvollständige oder überzählige Eingaben reagieren soll. Darüber gibt es zwischen den Entwicklern von Programmiersprachen keine Einigkeit.

t.Zero wartet bei einer unvollständigen Eingabe auf weiteren Input. Man darf einen Eingabeausdruck über mehrere Zeilen verteilen. Bei Funktionsdefinitionen haben wir das routinemäßig getan. Ein Zuviel an Eingabe wird dagegen als Fehler angesehen:

```
-> 1 + 1
Error: Trailing input
```

Die Zeichenfolge `1 + 1` ist keine syntaktisch korrekte Eingabe. Um in t.Zero den Wert des mathematischen Ausdrucks `1 + 1` zu berechnen, müsste man `(+ 1 1)` eingeben. Weil die erste 1 in der Eingabe `1 + 1` selbst schon eine zulässige Eingabe ist, wird das anschließende `'+ 1'` als überzählig zurückgewiesen.

1.2 Beispiele

Wir haben jetzt alle wesentlichen Elemente der Sprache kennengelernt. Man kann mit diesen wenigen Mitteln erstaunlich viel erreichen.

Vorweg noch eine Bemerkung zur Art des Arbeitens mit t.Zero. Es ist auf die Dauer nicht sinnvoll, Definitionen bei jedem Aufruf des Interpreters von Neuem einzugeben. Besser ist es, sie in eine Datei zu schreiben, aus der sie dann eingelesen werden. Die obige Definition der Funktion `factorial` könnte zum Beispiel in der Datei `factorial.tzero` stehen. Die Endung des Dateinamens darf beliebig gewählt werden.

Den Namen der Datei, aus der man die Eingaben lesen will oder, falls gewünscht, auch die Namen mehrerer Dateien, übergibt man t.Zero beim Aufruf. Wenn `factorial.tzero` die Definition der Funktion `factorial` und den Ausdruck `(factorial 10)` enthält, sieht das so aus:

```
$ tzero factorial.tzero
*** tZero ***
To quit, type '(quit)'
function[factorial]
3628800
->
```

Der Interpreter zeigt zu jedem Ausdruck in der Datei das Ergebnis der Auswertung an. Anschließend erscheint der Prompt und man kann interaktiv weiterarbeiten.

Es empfiehlt sich, jede gespeicherte Funktionsdefinition zu kommentieren. In *t.Zero* und allen anderen Programmiersprachen dieses Buchs beginnen Kommentare mit einem Semikolon und reichen bis zum Zeilenende.

1.2.1 Zahlenspielereien

Das Einmaleins von $n = 142857$ hat die seltsame Eigenschaft, dass die ersten sechs Vielfachen von n alle durch zyklische Vertauschung der Ziffern von n entstehen.

Wir speichern n als argumentlose Funktion und prüfen das nach:

```
-> (define (n) 142857)
function[n]
-> (define (multiple i) (* (n) i))
function[multiple]
-> (multiple 2)
285714
-> (multiple 3)
428571
-> (multiple 4)
571428
-> (multiple 5)
714285
-> (multiple 6)
857142
```

Es gibt nur sechs zyklische Vertauschungen bei einer sechsstelligen Zahl. Bei `(multiple 7)` muss das Resultat deshalb anders aussehen:

```
-> (multiple 7)
999999
```

Wie geht es weiter?

```
-> (multiple 8)
1142856
```

Das ist wieder fast genau die Zahl n , nur hat sich die letzte Ziffer 7 aufgespalten in die 6 und eine 1, die nach vorne gewandert ist. Man kann jetzt ahnen, wie es weitergeht. Probieren Sie es aus.

Gibt es noch weitere Zahlen n mit einem ähnlichen Verhalten? Aus $7n = 999999$ folgt, dass n die Periode von $1/7 = 0.142857\dots$ ist. Es gibt tatsächlich weitere Zahlen p mit der Eigenschaft, dass die Vielfachen der Periode n von $1/p$ durch zyklische Vertauschung von n entstehen. Kleine Primzahlen sind gute Kandidaten für p . Experimentieren Sie!

1.2.2 Größter gemeinsamer Teiler

Der vermutlich erste Algorithmus aller Zeiten ist der Euklidische Algorithmus zur Berechnung des größten gemeinsamen Teilers (Gcd, greatest common divisor) von zwei natürlichen Zahlen m und n .

Der Euklidische Algorithmus wird von t.Zero jedes Mal benutzt, wenn ein Bruch eingegeben oder als Resultat einer Rechnung erzeugt wird. Bei der Eingabe

```
-> 63/28
9/4
```

werden Zähler und Nenner mit dem Gcd 7 der beiden Zahlen gekürzt.

Der Gcd von zwei natürlichen Zahlen $m \geq n$ wird nach Euklid folgendermaßen berechnet: Im Fall $n = 0$ ist der Gcd m , sonst schreibt man $m = an + r$ mit $a = \lfloor m/n \rfloor$ und dem Rest $r = m - an$ und berechnet den Gcd von n und r .

Das kann man unmittelbar als rekursive t.Zero-Funktion formulieren (dargestellt ist der Inhalt der Datei ggt.tzero, deshalb erscheint kein Prompt):

```
; Modulo-Funktion
(define (mod m n) (- m (* n (trunc (/ m n)))))

; gcd von m und n
(define (gcd m n)
  (if (= n 0) m
      (gcd n (mod m n))))
```

Wir probieren die Funktion aus:

```
-> (trace gcd)
Tracing mode for function[gcd] is on
-> (gcd 28 63)
Call (gcd 28 63)
Call (gcd 63 28)
Call (gcd 28 7)
Call (gcd 7 0)
Return 7 from (gcd 7 0)
Return 7 from (gcd 28 7)
Return 7 from (gcd 63 28)
Return 7 from (gcd 28 63)
7
```

Die Ausdrücke (gcd 28 63), (gcd 63 28), (gcd 28 7) und (gcd 7 0), die dabei erzeugt und ausgewertet werden, liefern alle dasselbe Resultat 7: Die Funktion gcd ist *endrekursiv*, das heißt, das Resultat des letzten Aufrufs ist auch schon der Wert des ursprünglichen Aufrufs.

In einer imperativen Programmiersprache würde man in einem Programm zur Berechnung des Gcd vielleicht zwei lokale Variablen a und r deklarieren und eine While-Schleife verwenden. In t.Zero ist das nicht möglich, weil es keine Variablen und keine Schleifen gibt. Die obige Definition ist aber gerade deshalb besonders kompakt und übersichtlich.

1.2.3 Quadratwurzel

Heronsches Verfahren

Die Quadratwurzel aus einer Zahl $a > 0$ kann man mit einem nach Heron von Alexandria benannten Verfahren schnell mit beliebiger Genauigkeit berechnen. Es ist ein Spezialfall des Newton-Verfahrens zur Berechnung von Nullstellen.

Man startet mit einem Näherungswert $x \neq 0$ für \sqrt{a} und berechnet daraus den neuen, besseren Näherungswert

$$x' = \frac{1}{2} \left(x + \frac{a}{x} \right).$$

Als erste Näherung kann man jedes beliebige $x \neq 0$ wählen; damit berechnet man iterativ immer bessere Näherungen. Sobald man eine Näherung gefunden hat, bei der mindestens eine Nachkommastelle richtig ist, verdoppelt sich in jedem weiteren Schritt die Zahl der gültigen Stellen.

In *t.Zero* formuliert:

```
; Ein Schritt beim Heron-Verfahren für die Wurzel aus a > 0
; x ist eine Ausgangsnäherung
(define (step a x)
  (* 1/2 (+ x (/ a x))))

; n Iterationen von step
(define (iterate a x n)
  (if (= n 0) x
      (step a (iterate a x (- n 1)))))

; Berechnung der Wurzel aus a, Startwert 1, n Schritte
(define (sqrt a n)
  (iterate a 1 n))
```

Beim Ausprobieren sieht man, wie die Genauigkeit schnell wächst:

```
-> (round (sqrt 9 4) 20)
3.00009155413138017853
-> (round (sqrt 9 5) 20)
3.00000000139698386224
-> (round (sqrt 9 6) 20)
3.00000000000000000032
```

Wir hätten die Funktion *iterate* offensichtlich auch in der folgenden Form definieren können:

```
; n Iterationen von step, endrekursive Version
(define (Iterate a x n)
  (if (= n 0) x
      (Iterate a (step a x) (- n 1))))

; Berechnung der Wurzel aus a mit endrekursiver Iteration
(define (Sqrt a n)
  (Iterate a 1 n))
```

Die Werte, die man mit *Sqrt* erhält, unterscheiden sich nicht von denen der Funktion *sqrt*. Trotzdem verläuft die Rechnung ganz anders:

```
-> (trace step iterate Iterate)
Tracing mode for function[step] is on
Tracing mode for function[iterate] is on
Tracing mode for function[Iterate] is on
```

```

-> (sqrt 9 2)
      Call (iterate 9 1 2)
      Call (iterate 9 1 1)
      Call (iterate 9 1 0)
      Return 1 from (iterate 9 1 0)
      Call (step 9 1)
      Return 5 from (step 9 1)
      Return 5 from (iterate 9 1 1)
      Call (step 9 5)
      Return 17/5 from (step 9 5)
      Return 17/5 from (iterate 9 1 2)

17/5
-> (Sqrt 9 2)
      Call (Iterate 9 1 2)
      Call (step 9 1)
      Return 5 from (step 9 1)
      Call (Iterate 9 5 1)
      Call (step 9 5)
      Return 17/5 from (step 9 5)
      Call (Iterate 9 17/5 0)
      Return 17/5 from (Iterate 9 17/5 0)
      Return 17/5 from (Iterate 9 5 1)
      Return 17/5 from (Iterate 9 1 2)

17/5

```

Im ersten Fall werden alle rekursiven Aufrufe von `iterate` begonnen, bevor der erste Aufruf der Funktion `step` erfolgt. Bei der zweiten Version werden die Aufrufe von `step` schon „auf dem Hinweg“ erledigt, also während die rekursiven Aufrufe von `Iterate` begonnen werden. Die Rückabwicklung der Rekursion ist in diesem Fall ziemlich uninteressant, der Rückgabewert ist immer derselbe.

Der Grund ist – wie bei der Berechnung des Gcd im vorigen Abschnitt – die Endrekursivität einer Funktion: Der rekursive Aufruf von `Iterate` ist die *letzte* Aktion bei der Auswertung des Rumpfs der Funktion `Iterate`.

Im Abschnitt 4.6 werden wir sehen, wie man die unproduktive Rückabwicklung von endrekursiven Funktionen wegoptimieren kann.

Ganzzahlige Quadratwurzel

Die obige Implementierung der Quadratwurzelberechnung hat zwei Nachteile, die sich mit wenig Aufwand beheben lassen:

1. Der Startwert 1 ist für große a zu grob gewählt. Es wäre besser, mit einer Näherung zu beginnen, die ungefähr halb so viele Dezimalstellen vor dem Komma hat wie a .
2. `t.Zero` rechnet immer exakt. Deshalb sollte für Argumente der Form $a = b^2$ auch wirklich die genaue Wurzel $b > 0$ gefunden werden.

Beide Probleme kann man lösen, indem man eine Funktion `isqrt` definiert, die $\lfloor \sqrt{a} \rfloor$, also die größte ganze Zahl n mit $n \leq \sqrt{a}$ berechnet.

Zu diesem Zweck ersetzen wir die Funktion `iterate` durch eine Funktion `iterate-truncated`, in der nach jedem Schritt die Nachkommastellen abgeschnitten werden:

```

; Iteriere step mit Abschneiden der Nachkommastellen,
; bis das Resultat <= sqrt(a) ist
(define (iterate-truncated a x)
  (if (<= (* x x) a) x
      (iterate-truncated a (trunc (step a x)))))

```

Wenn man sicherstellt, dass die Iteration mit einem Startwert begonnen wird, der größer ist als $\lfloor \sqrt{a} \rfloor$, dann nähern sich die Zwischenschritte von oben der Zahl $\lfloor \sqrt{a} \rfloor$ und erreichen sie nach wenigen Schritten:

```
; Rate eine nicht zu große Zahl m mit sqrt(a) < m
(define (guess k a)
  (if (< a (* k k)) k
      (guess (* 10 k) a)))

; Ganzzahlige Quadratwurzel von a
(define (isqrt a) (iterate-truncated a (guess 1 a)))
```

Die Funktionen `iterate-truncated` und `guess` sind endrekursiv.

Es ist erstaunlich, wie genau diese Methode arbeitet:

```
-> (isqrt 100)
10
-> (isqrt 99.999999999999)
9
```

Damit können wir nun die beiden obigen Nachteile der Funktion `sqrt` loswerden.

Das erste Problem ist sofort behoben: Als Startwert für die näherungsweise Berechnung der Quadratwurzel von a wird `(isqrt a)` gewählt.

Um für Brüche, die Quadratzahlen sind, die Wurzel exakt zu ermitteln, definieren wir mit der Hilfe von `isqrt` eine weitere Funktion `rational-sqrt`, die zu einem Bruch $\frac{a}{b} > 0$ die größte ganze Zahl m mit $\frac{m}{b} \leq \sqrt{\frac{a}{b}}$ berechnet. Das ist, wie man leicht sieht, $\frac{1}{b} \cdot \lfloor \sqrt{ab} \rfloor$.

Mit den Zugriffoperatoren `num` und `den` für Zähler und Nenner einer rationalen Zahl kann man `rational-sqrt` so ausdrücken:

```
; Rationale Quadratwurzel aus x > 0. Ausgabe als Bruch.
(define (rational-sqrt x)
  (/ (isqrt (* (num x) (den x))) (den x)))

; Rationale Quadratwurzel aus x > 0 mit Dezimaldarstellung, falls möglich.
(define (rat-sqrt x)
  (* 1.0 (rational-sqrt x)))
```

Wir probieren die neuen Funktionen aus:

```
-> (rational-sqrt 729/1369)
27/37
-> (square 0.123456789)
0.015241578750190521
-> (rat-sqrt %)
0.123456789
```

Mit dem Startwert `(rational-sqrt a)` anstelle von `(isqrt a)` konvergiert das Heron-Verfahren in der Regel noch deutlich schneller.

1.2.4 Primzahlen

Die einfachste Art zu entscheiden, ob die natürliche Zahl n eine Primzahl ist, besteht darin, für alle Zahlen m , $2 \leq m \leq \sqrt{n}$ zu prüfen, ob m Teiler von n ist. Die Funktion `isqrt` aus dem vorigen Beispiel zur Berechnung der ganzzahligen Quadratwurzel ist dabei von Nutzen.

Das „für alle $m \leq \sqrt{n}$ “ wird durch eine rekursive Funktion ausgedrückt: (factor-in-range? x y n) testet, ob eine der Zahlen $x, x+2, x+4, \dots, y$ ein Teiler von n ist. Damit diese Funktion nur die ungeraden Zahlen testen muss, wird in der Funktion prime? zuerst abgefragt, ob n gerade ist:

```
;   Ganzzahlige Division
(define (div m n)
  (trunc (/ m n)))

;   Ist x ein Teiler von n?
(define (factor? x n)
  (= n (* x (div n x))))

;   Ist eine der Zahlen im Bereich x, x+2, x+4, ... ,y ein Teiler von n?
(define (factor-in-range? x y n)
  (if (< y x) (false)
      (if (factor? x n) (true)
          (factor-in-range? (+ x 2) y n))))

;   Ist n eine Primzahl?
(define (prime? n)
  (if (= n 2) (true)
      (if (factor? 2 n) (false)
          (not (factor-in-range? 3 (isqrt n) n)))))
```

Anwendungsbeispiel:

```
-> (prime? 1000003)
true
```

Namen von Funktionen, die ein boolesches Resultat haben, enden oft mit einem Fragezeichen. Das ist eine Konvention, der man nicht in jedem Fall folgen muss: '(> x y)' ist verständlicher als '(greater? x y)'.

1.2.5 Potenzen

In t.Zero wird fast alles rekursiv programmiert. Nur sehr einfache Funktionen kommen ohne dieses Hilfsmittel aus.

Rekursion steht manchmal in dem Verdacht, ein komplizierteres Konzept zu sein als die allseits beliebte Schleife. Dieses Vorurteil hat seinen Grund wohl darin, dass die meisten von uns als Erstes die Programmierung mit Hilfe von Schleifen kennengelernt haben. Tatsächlich ist Rekursion in vielen Fällen die natürlichste Art, einen Ablauf zu beschreiben.

Das Beispiel der ganzzahligen Potenzen einer Zahl x illustriert das und zeigt zugleich, warum die einfachste Art, etwas rekursiv auszudrücken, manchmal nicht die beste ist.

Die n -te Potenz von x ist $x^n = x \cdot \dots \cdot x$ mit n Faktoren x . Will man genau sagen, was „ \dots “ bedeutet, so muss man formaler werden:

$$x^n = \begin{cases} 1 & \text{falls } n = 0, \\ x \cdot x^{n-1} & \text{falls } n > 0. \end{cases}$$

Das lässt sich fast Wort für Wort in eine rekursive Funktion übertragen:

```
;   n-te Potenz von x, n >= 0 ganzzahlig
(define (power x n)
  (if (= n 0) 1 ; 1, falls n = 0
      (* x (power x (- n 1))))) ; x * x^(n - 1) sonst
```

Die Laufzeit dieser Funktion ist proportional zu n . Das ist für die Praxis – selbst bei unseren bescheidenen Ansprüchen an die Rechengeschwindigkeit – zu langsam.

Und schon sieht man ein typisches Problem rekursiver Programmierung: Eine kompliziertere Formulierung desselben Sachverhalts führt leider oft zu effizienteren Programmen.

Für $n > 0$ gilt nämlich auch die Beziehung

$$x^n = \begin{cases} 1 & \text{falls } n = 0, \\ (x^{n/2})^2 & \text{falls } n \text{ gerade,} \\ x \cdot (x^{\lfloor n/2 \rfloor})^2 & \text{falls } n > 0 \text{ ungerade.} \end{cases}$$

In *t.Zero*-Notation ergibt das die folgende Funktion:

```
; Ist n gerade?
(define (even? n)
  (= 0 (mod n 2)))

; Potenzen von x, n >= 0 und ganzzahlig
(define (Power x n)
  (if (= n 0) 1
      (if (even? n) (square (Power x (/ n 2)))
          (* x (square (Power x (/ (- n 1) 2)))))))
```

Das Programm sieht nicht mehr ganz so einfach aus, aber der Gewinn an Effizienz gegenüber der ersten Version ist dramatisch. Bei jedem rekursiven Aufruf halbiert sich nun der Exponent n , es finden nur noch etwa $\log_2 n$ rekursive Aufrufe statt.

Im Jahr 1876 hat der französische Mathematiklehrer Édouard Lucas die größte jemals von Hand berechnete Primzahl ermittelt: $p = 2^{127} - 1$. Wir verfolgen die Berechnung von p im Tracing-Modus:

```
-> (trace Power)
Tracing mode for function[Power] is on
-> (- (Power 2 127) 1)
  Call    (Power 2 127)
  Call    (Power 4 63)
  Call    (Power 16 31)
  Call    (Power 256 15)
  Call    (Power 65536 7)
  Call    (Power 4294967296 3)
  Call    (Power 18446744073709551616 1)
  Call    (Power 340282366920938463463374607431768211456 0)
  Return  1 from (Power 340282366920938463463374607431768211456 0)
  Return  18446744073709551616 from (Power 18446744073709551616 1)
  Return  79228162514264337593543950336 from (Power 4294967296 3)
  Return  5192296858534827628530496329220096 from (Power 65536 7)
  Return  1329227995784915872903807060280344576 from (Power 256 15)
  Return  21267647932558653966460912964485513216 from (Power 16 31)
  Return  85070591730234615865843651857942052864 from (Power 4 63)
  Return  170141183460469231731687303715884105728 from (Power 2 127)
170141183460469231731687303715884105727
```

Von 127 Aufrufen der Funktion *power* sind gerade mal acht übrig geblieben. Die natürliche rekursive Formulierung ist offenbar weit weniger geeignet als die weniger natürliche Variante. Wer allerdings glaubt, solchen Problemen mit rekursiven Programmen dadurch zu entkommen, dass er bei der guten alten Programmierung mit Schleifen bleibt, der täuscht sich. Selbstverständlich kann man die beiden Algorithmen zur Berechnung von x^n auch ohne Rekursion programmieren – aber die rekursionsfreie schnelle Version sieht keineswegs einfacher aus als *Power*.

1.2.6 Berechnung von π

Die Formel von Borwein-Bailey-Plouffe

Der Zweck der ersten Computer war fast ausschließlich das „number crunching“ – die Berechnung von Logarithmen, trigonometrischen Funktionen, Zinstabellen etc. Für solche Zwecke ist selbst eine so rudimentäre Programmiersprache wie t.Zero ganz gut geeignet.

Das nachfolgende t.Zero-Programm zur Berechnung der Dezimaldarstellung der Zahl π demonstriert das. Es beruht auf einer Formel für π , die 1995 von D. Bailey, P. Borwein und S. Plouffe entdeckt wurde ([3]):

$$\pi = \sum_{i=0}^{\infty} \frac{1}{16^i} \cdot \left(\frac{4}{8i+1} - \frac{2}{8i+4} - \frac{1}{8i+5} - \frac{1}{8i+6} \right).$$

Diese Formel stellte zum Zeitpunkt ihrer Entdeckung eine kleine Sensation dar, weil sie dazu verwendet werden kann, einzelne Stellen der Hexadezimalentwicklung von π ohne die Kenntnis aller vorangehenden Stellen zu berechnen. Das galt bis dahin als unmöglich. Für die Dezimalentwicklung von π kennt man eine solche Formel bis heute nicht.

Wir müssen die Summanden a_i und die Partialsummen $s_n = a_0 + \dots + a_n$ berechnen:

```
; i-ter Summand
(define (summand i)
  (/ (- (-
        (/ 4 (+ (* 8 i) 1))
        (/ 2 (+ (* 8 i) 4)))
      (/ 1 (+ (* 8 i) 5)))
     (/ 1 (+ (* 8 i) 6)))
    (power 16 i)))

; n-te Partialsumme
(define (partialsum n)
  (if (= n 0) (summand 0)
      (+ (partialsum (- n 1)) (summand n))))
```

Wie groß muss n sein? Eine ziemlich einfache Abschätzung, auf deren Herleitung wir hier verzichten, besagt, dass für N -stellige Genauigkeit die Partialsumme S_N ausreicht. (Mit etwas mehr Aufwand kann man beweisen, dass bereits $\approx 0.85 \cdot N$ Summanden genügen.)

```
; Berechnung von pi auf N Nachkommastellen genau
(define (pi N)
  (round (partialsum N) N))
```

Ludolph van Ceulen (1540–1610) hat an der Berechnung der ersten 35 Dezimalstellen von π angeblich fast dreißig Jahre lang gearbeitet. Mit t.Zero geht es deutlich schneller:

```
-> (pi 35)
3.14159265358979323846264338327950288
```

Die Rechenzeit hat sich von dreißig Jahren auf ungefähr ebenso viele Millisekunden verringert. Überdies ist das obige Programm *sehr* viel einfacher als van Ceulens Methode (er hat den Kreis durch ein gleichseitiges 62-Eck angenähert).

Verbesserung durch Runden und Endrekursion

Das eben entwickelte Programm für π ist in gewisser Hinsicht optimal: Es ist kurz, korrekt und leicht zu verstehen. Leider ist es nicht besonders schnell und obwohl Effizienz nicht zu den Entwurfszielen von *t.Zero* gehört, soll noch gezeigt werden, wie man mit einfachen Mitteln eine Verbesserung der Laufzeit um einen Faktor 3 bis 4 erreichen kann.

Die Summanden der Formel von Bailey-Borwein-Plouffe werden schnell klein – genauer gesagt: Die Nenner werden groß. Ein Beispiel:

```
-> (summand 50)
307597/5350765772415800615784641425997483505021064587840152526962210730147840
```

Die Addition von Brüchen mit großen Zählern oder Nennern ist relativ aufwendig. Deshalb ist es besser, nicht mit exakten, sondern mit gerundeten Summanden zu rechnen:

```
; i-ter Summand a_i, auf N Stellen gerundet
(define (rounded-summand i N)
  (round (summand i) N))
```

Von diesen sollen nur so viele wie nötig aufsummiert werden. Der Schlüssel dazu ist die Umformulierung von *partialsum* in eine endrekursive Funktion:

```
; Endrekursive Berechnung der Summe mit auf N Stellen gerundeten Summanden
(define (rounded-partialsum s x i N)
  (if (= x 0) s
      (rounded-partialsum (+ s x) (rounded-summand (+ i 1) N) (+ i 1) N)))

; pi auf N Dezimalstellen genau, mit Rundungsfehlern
(define (rounded-pi N)
  (rounded-partialsum 0 (rounded-summand 0 N) 0 N))
```

Was dabei im Detail abläuft, sieht man im Tracing-Modus:

```
-> (trace rounded-partialsum)
Tracing mode for function[rounded-partialsum] is on
-> (rounded-pi 8)
Call (rounded-partialsum 0 3.13333333 0 8)
Call (rounded-partialsum 3.13333333 0.00808913 1 8)
Call (rounded-partialsum 3.14142246 0.00016492 2 8)
Call (rounded-partialsum 3.14158738 0.00000506 3 8)
Call (rounded-partialsum 3.14159244 0.00000018 4 8)
Call (rounded-partialsum 3.14159262 0.0 5 8)
Return 3.14159262 from (rounded-partialsum 3.14159262 0.0 5 8)
Return 3.14159262 from (rounded-partialsum 3.14159244 0.00000018 4 8)
Return 3.14159262 from (rounded-partialsum 3.14158738 0.00000506 3 8)
Return 3.14159262 from (rounded-partialsum 3.14142246 0.00016492 2 8)
Return 3.14159262 from (rounded-partialsum 3.13333333 0.00808913 1 8)
Return 3.14159262 from (rounded-partialsum 0 3.13333333 0 8)
3.14159262
```

Die Funktion *rounded-partialsum* ruft sich selbst so lange mit wachsender Partialsumme *s* und kleiner werdendem Summanden *x* auf, bis der Summand (*rounded-summand* 5 8) den Wert 0.0 ergibt. Das Argument 3.14159262 dieses letzten Aufrufs wird als Resultat zurückgegeben. Der Trick würde mit exakten Summanden nicht funktionieren, weil diese immer von Null verschieden sind.

Weil mit gerundeten Summanden gerechnet wird, hat die Summe einen Fehler an der letzten Stelle, für größere *N* sogar an den letzten zwei oder drei Stellen. Um diesen Fehler zu vermeiden,

rechnen wir für die engültige Version des Programms intern mit ein paar zusätzlichen Stellen und schneiden die Extrastellen hinterher wieder ab:

```
; Berechnung von pi auf N Dezimalstellen mit 5 internen Zusatzstellen.
; Achtung: (Pi n) und (pi n) arbeiten intern unterschiedlich!
(define (Pi N)
  (round (rounded-pi (+ N 5)) N))
```

Der Letzte, der π von Hand berechnet hat, war William Shanks (1812–1882). Er hat von etwa 1840 an bis 1873 die ersten 707 Stellen berechnet. 1945 stellte sich heraus, dass davon nur die ersten 527 Stellen korrekt waren. Der berichtigte Wert sieht so aus:

```
-> (Pi 707)
3.141592653589793238462643383279502884197169399375105820974944592307816406286
20899862803482534211706798214808651328230664709384460955058223172535940812848
11174502841027019385211055596446229489549303819644288109756659334461284756482
33786783165271201909145648566923460348610454326648213393607260249141273724587
00660631558817488152092096282925409171536436789259036001133053054882046652138
41469519415116094330572703657595919530921861173819326117931051185480744623799
62749567351885752724891227938183011949129833673362440656643086021394946395224
73719070217986094370277053921717629317675238467481846766940513200056812714526
35608277857713427577896091736371787214684409012249534301465495853710507922796
8925892354201995
```

Der Aufruf von `(pi 707)` liefert dasselbe Resultat, dauert aber merklich länger.

Trotzdem ist auch das optimierte Programm alles andere als schnell. Der t.Zero-Interpreter ist langsam und der verwendete Algorithmus ist kein bisschen effizient. Es gibt viel bessere Verfahren.

1.2.7 Exponentialfunktion

Die Berechnung von Exponentialfunktion, Logarithmus, trigonometrischen und ähnlichen Funktionen funktioniert grundsätzlich nach demselben Prinzip wie die Berechnung von π . Man summiert hinreichend viele Summanden einer möglichst gut konvergierenden Reihe auf, im einfachsten Fall beispielsweise nach dem folgenden Schema:

```
; Partialsumme einer Reihe an der Stelle x bis zum n-ten Summanden
(define (series x n)
  (if (< n 0) 0 (+ (series x (- n 1)) (summand x n))))
```

Je nach Wahl von `summand` erhält man die gewünschte Reihe.

Die Zahl e

Die Reihe der Exponentialfunktion konvergiert sehr gut:

$$\exp(x) = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

Formulierung mit t.Zero:

```
; n-ter Summand der Exponentialfunktion an der Stelle x
(define (exp-summand x n)
  (/ (power x n) (factorial n)))
```

```

; n-te Partialsumme der Exponentialreihe an der Stelle x
(define (exp-series x n)
  (if (< n 0) 0
      (+ (exp-series x (- n 1)) (exp-summand x n))))

```

An der Stelle $x = 1$ genügen N Summanden für N Stellen Genauigkeit. Um ganz sicherzugehen, rechnen wir mit einem zusätzlichen Summanden:

```

; e^x auf N Stellen genau, 1. Version
; Nur gut für x <= 1, sonst zu wenige Summanden
(define (exp-version1 x N)
  (round (exp-series x (+ N 1)) N))

; Eulersche Zahl auf N Dezimalstellen
(define (euler N)
  (exp-version1 1 N))

```

Ausprobieren:

```

-> (euler 50)
2.71828182845904523536028747135266249775724709369995

```

Steuerung der Genauigkeit

Für $x \neq 1$ ist es nicht so einfach abzuschätzen, wie viele Summanden man für eine gewünschte Genauigkeit braucht.

Das hängt von x und von der jeweiligen Reihe ab. Am naheliegendsten ist es, so wie in Abschnitt 1.2.6 die Summation dann zu beenden, wenn die auf N Stellen berechneten Summanden null werden.

Das Schema der rekursiven Formulierung einer Summation mit Abbruchbedingung sieht etwas anders aus als das obige Schema zur Reihenberechnung:

```

; Reihe auf N Stellen genau
; s ist die Summe der Reihe an der Stelle x bis zum (n-1)-ten Summanden
(define (series x s n N)
  (if (= 0 (summand x n N)) s
      (series x (+ s (summand x n N)) (+ n 1) N)))

```

Bei jedem Aufruf von `series` werden das Argument x , die $(n-1)$ -te Partialsumme s und der Index n als Parameter übergeben. N ist die gewünschte Stellenzahl.

Für die Exponentialreihe bedeutet das:

```

; Auf N Stellen gerundeter Summand
(define (rounded-exp-summand x n N)
  (round (exp-summand x n) N))

; Exponentialreihe mit gerundeten Summanden
(define (rounded-exp-series x s n N)
  (if (= 0 (rounded-exp-summand x n N)) s
      (rounded-exp-series x (+ s (rounded-exp-summand x n N)) (+ n 1) N)))

; e^x auf N Stellen genau, 2. Version
(define (exp-version2 x N)
  (round (rounded-exp-series x 0 0 (+ N 5)) N))

```

Der Vergleich von `(exp-version2 5 20)` mit e^5 , auf 20 Stellen gerundet, gibt Zutrauen in die Korrektheit der Funktion `exp-version2`:

```

-> (exp-version2 5 20)
148.41315910257660342111
-> (round (power (euler 25) 5) 20)
148.41315910257660342111

```

Allerdings wird bei dieser Version jeder Summand zweimal berechnet: einmal für die Abbruchbedingung und dann nochmal für die Summation.

Die einfachste Lösung wäre es, den doppelt vorkommenden Ausdruck nur einmal zu berechnen und das Ergebnis in einer Variablen zu speichern. Das ist in t.Zero nicht möglich – es gibt keine Variablen.

Man kann das Problem aber umgehen, indem man nicht nur die bisher errechnete Summe, sondern auch den nächsten Summanden in jedem Rekursionsschritt weitergibt:

```

; Exponentialreihe auf N Stellen genau, 3. Version
; s ist die Summe der Reihe bis zum (n-1)-ten Summanden
; a ist der n-te Summand
(define (exp-series-version3 x s a n N)
  (if (= 0 a) s
      (exp-series-version3 x (+ s a)
                           (rounded-exp-summand x (+ n 1) N) (+ n 1) N)))

; e^x auf N Stellen genau, 3. Version
; Rechnung mit 5 Extrastellen
(define (exp-version3 x N)
  (round (exp-series-version3 x 0
                              (rounded-exp-summand x 0 (+ N 5)) 0 (+ N 5)) N))

```

Jetzt wird jeder Summand nur noch einmal berechnet.

Beschleunigung der Konvergenz

Das Ergebnis ist immer noch nicht ganz zufriedenstellend. Für Werte von x , die deutlich größer als 1 sind, konvergiert die Exponentialreihe immer langsamer, es müssen zu viele Summanden berechnet werden.

Dem kann man abhelfen, indem man x in seinen ganzzahligen Anteil n und die Nachkommastellen y zerlegt: $x = n + y$. Damit berechnet man dann $e^x = e^n \cdot e^y$ in zwei Teilen: Für e^n wird die Funktion `power` benutzt, für e^y die Funktion `exp`.

Die Idee in t.Zero umgesetzt ergibt eine vierte Variante der Exponentialfunktion. Der Code sieht weniger elegant aus als der bisherige, ist aber viel effizienter:

```

; Ganzzahliger Anteil von x
(define (integer-part x)
  (round x 0))

; Gebrochener Anteil von x
(define (fractional-part x)
  (- x (integer-part x)))

; e^n * e^y auf N Stellen genau
(define (exp-product n y N)
  (* (power (exp-version3 1 N) n) (exp-version3 y N)))

```

```

; e^x auf N Stellen genau, 4. Version
(define (exp-version4 x N)
  (round
    (if (< x 1)
      (exp_version3 x (+ N 5))
      (exp-product (integer-part x) (fractional-part x) (+ N 5)))
    N))

; e^x auf N Stellen genau, mit anderem Namen
(define (exp x N)
  (exp-version4 x N))

```

Wer an weiteren Verbesserungen von Genauigkeit und Effizienz interessiert ist, findet hier noch ein reiches Betätigungsfeld.

Man kann zum Beispiel x in der Form $x = n \ln(2) + y$ mit ganzzahligem $n = \lfloor x/\ln(2) \rfloor$ schreiben. Dann ist $0 \leq y < \ln(2) = 0.69 \dots$ und $e^x = 2^n \cdot e^y$. Ganzzahlige Potenzen von 2 lassen sich leichter berechnen als von e und die Reihe für e^y konvergiert schneller als bei dem obigen Ansatz, bei dem nur $y < 1$ gesichert ist.

Eine weitere Möglichkeit besteht darin, die Reihe für e^x so zu schreiben, dass keine Potenzen von x explizit zu berechnen sind:

$$e^x = \frac{1}{0!} + x \cdot \left(\frac{1}{1!} + x \cdot \left(\frac{1}{2!} + x \cdot \left(\frac{1}{3!} + \dots \right) \right) \right).$$

Das lädt zu einer rekursiven Formulierung ein. In Kombination mit Zerlegungen der Form $e^x = a^n \cdot e^y$ kann man damit sehr effiziente Programme zur Berechnung der Exponentialfunktion entwickeln.

1.2.8 Logarithmus

Den natürlichen Logarithmus kann man mit der folgenden, für alle $x > 0$ konvergenten Reihe berechnen:

$$\ln(x) = 2 \cdot \sum_{n=0}^{\infty} \frac{1}{2n+1} \cdot \left(\frac{x-1}{x+1} \right)^{2n+1}.$$

Mit den Mitteln des vorigen Abschnitts können wir sie leicht in Programmcode übersetzen:

```

; Gerundete Division mit N Nachkommastellen
(define (// x y N)
  (round (/ x y) N))

; n-ter Summand an der Stelle z, gerundet auf N Nachkommastellen
(define (lnsummand z n N)
  (// (power z (+ 1 (* 2 n))) (+ 1 (* 2 n)) N))

; Reihe an der Stelle z mit gerundeten Summanden, N Nachkommastellen
; s ist die Summe der Reihe bis zum (n-1)-ten Summanden
; a ist der n-te Summand
(define (lnseries s a n z N)
  (if (= a 0) s
      (lnseries (+ s a) (lnsummand z (+ n 1) N) (+ n 1) z N)))

; Reihe an der Stelle z vom 0-ten Summanden an, N Nachkommastellen
(define (lnseries z N)
  (lnseries 0 (lnsummand z 0 N) 0 z N))

```

```

; Argument der Reihe an der Stelle x, gerundet auf N Nachkommastellen
(define (arg x N)
  (// (- x 1) (+ x 1) N))

; Logarithmus von x > 0, N Nachkommastellen
(define (Ln x N)
  (round (* 2 (Lnseries (arg x (+ N 5)) (+ N 5))) N))

```

Die Funktion `lnseries` berechnet endrekursiv die Reihe. Statt $(x-1)/(x+1)$ wird z geschrieben. `Lnseries` (mit großem L) definiert die Reihe vom 0-ten Term an. In der Funktion `Ln` wird $(x-1)/(x+1)$ für z eingesetzt.

Die Reihe konvergiert gut, wenn x in der Nähe von 1 liegt. Für einigermaßen große und kleine Werte von x braucht man aber viel zu viele Summanden. Man kann das beobachten, wenn man für `lnseries` den Tracing-Modus aufruft. Der Grund ist klar: $(x-1)/(x+1)$ wird umso kleiner, je näher x an 1 liegt. Dann werden die Summanden schnell klein. Schon für mäßig große bzw. kleine x hat $(x-1)/(x+1)$ Werte in der Nähe von 1, mit der Folge, dass die verwendete Reihe immer mehr der Reihe $\sum_{n=0}^{\infty} 1/(2n+1)$ ähnelt, die bekanntlich gegen ∞ geht.

Wie bei der Exponentialfunktion ist es deshalb sinnvoll, eine Skalierung anzuwenden: Man schreibt $x = 2^n \cdot (x/2^n)$ bzw. $\ln(x) = n \ln(2) + \ln(x/2^n)$ und wählt n so, das $x/2^n$ möglichst nahe an 1 liegt:

```

; Ganzzahliger Teil des Zweierlogarithmus von x
(define (int-ld x)
  (if (< x 2) 0
      (+ 1 (int-ld (// x 2 0)))))

```

Für die Berechnung von $n \ln(2) + \ln(x/2^n)$ wird $\ln(2)$ als Konstante gespeichert. Das ist nicht unbedingt nötig, macht die Sache aber für größere N doch merklich schneller:

```

; Natürlicher Logarithmus von 2, 55 Nachkommastellen (berechnet mit (Ln 2 55))
(define (ln_of_2)
  0.6931471805599453094172321214581765680755001343602552541)

```

Der Rest ist einfach. Für $x < 1$ ersetzen wir $\ln(x)$ durch $-\ln(1/x)$ und rufen die skalierte Version des Logarithmus auf:

```

; Logarithmus von x > 0, N <= 55 Nachkommastellen
(define (ln x N)
  (if (< 55 N) (/ 1 0) ; gewaltsamer Abbruch bei N > 55
      (if (< x 1)
          (minus (ln (/ 1 x) N))
          (round (scaledLn x (int-ld x) (+ N 5)) N))))

; Negiertes x
(define (minus x)
  (- 0 x))

; Skalierter Logarithmus von x > 0, N Nachkommastellen,
; Logarithmus-Reihe ausgewertet an der Stelle x/(2^k) anstatt bei x
(define (scaledLn x k N)
  (+ (* k (round (ln_of_2) N))
      (Ln (/ x (power 2 k) N) N)))

```

Vorsicht ist angebracht: Weil $\ln(2)$ nur mit 55 Stellen gespeichert ist, sind die Resultate nur bis ungefähr $N = 50$ exakt. Bei zu großem N bricht die Funktion `ln` daher sicherheitshalber ab.

Dasselbe in Java:

```
// Abstand von (a, b) zum Nullpunkt
public static double distance(double a, double b) {
    return Math.sqrt(a*a + b*b);
}
```

Beides ist deklarativer Programmierstil, nur die Syntax ist anders (und im Fall von Java durch die Form des Ausdrucks $a*a + b*b$ den üblichen Lesegewohnheiten besser angepasst).

Der folgende Ausschnitt aus dem Handbuch eines Rechners vom Typ Zuse Z23 von 1962 (auch die Kommentare sind original) zeigt, wie man die Berechnung des Abstands im Formelcode, der Programmiersprache der Z23, schreiben musste:

```
C = A.A      Bildung von  $a^2 = c$ 
D = B.B      Bildung von  $b^2 = d$ 
D = D+C      Bildung von  $d = d + c = a^2 + b^2$ 
Z = WURZ D   Ziehen der Quadratwurzel
```

Der Formelcode war ein großer Fortschritt gegenüber dem Freiburger Code, der bis dahin gebräuchlichen Assembler-Notation der Zuse-Rechner. Trotzdem erlaubte er keine zusammengesetzten Ausdrücke, obwohl diese schon 1957 mit Fortran in die Programmierung Einzug gehalten hatten.

Im Formelcode *kann* man das Resultat nicht deklarativ ausdrücken. In Java ist das möglich, man wird es in der Regel auch tun, muss es aber nicht. Die Programmiersprache erlaubt es ebenso, wie beim Formelcode schrittweise mit Hilfsvariablen vorzugehen. t.Zero ist *rein deklarativ*, die Sprache kennt keine Variablen, keine Anweisungen und keine Befehlsfolgen. Eine solche Sprache zwingt uns eine ganz bestimmte Art der Programmierung auf.

Sehen wir uns nochmals die Exponentialfunktion aus Abschnitt 1.2.7 an. Die Definition ist nur wenige Zeilen lang:

```
; Summanden 0..n der Reihe für exp(x)
(define (exp-series x n)
  (if (< n 0) 0 (+ (exp-series x (- n 1)) (exp-summand x n))))

;  $x^n/n!$ 
(define (exp-summand x n)
  (/ (power x n) (factorial n)))
```

Diese Notation ist nicht nur ähnlich knapp wie die mathematische Definition der Reihenentwicklung der Exponentialfunktion, sie liefert auch ein Muster für andere Reihenberechnungen. Programmiersprachen haben einen sehr bestimmenden Einfluss auf die Art, wie man in ihnen Dinge zum Ausdruck bringt. Sie drängen uns in Denkmuster hinein, sie legen Lösungsansätze nahe oder verhindern sie – in gewisser Weise programmiert eine Programmiersprache das Denken derer, die diese Sprache benutzen.

Deklarative Programmiersprachen bringen uns dazu, weniger über den Weg zu einer Lösung nachzudenken und mehr darüber, was wir eigentlich mit „Lösung“ meinen. Darin steckt tendenziell eine Konzentration auf das Wesentliche, weg von überflüssigem Detail. Tatsächlich hat sich gezeigt, dass eine konsequent deklarative Programmierung weniger fehleranfällig ist als die klassische imperative Art, Programme zu schreiben, und mit geringerem Zeit- und Entwurfsaufwand zu korrekten Ergebnissen führt.

1.3.2 Probleme des deklarativen Programmierstils

A Lisp programmer knows the value of everything, but the cost of nothing. (A. Perlis, [35])

Warum programmiert angesichts der oben genannten Vorteile nicht alle Welt deklarativ? Dafür gibt es mehrere Gründe.

Der am häufigsten genannte Grund ist, dass deklarative Programmiersprachen weniger effizient sind als imperative Sprachen. Das Zitat von Alan Perlis, dem ersten Träger des Turing-Preises, formuliert diesen Einwand: Deklarative Programme, in denen Resultate durch Evaluation berechnet werden, sind in aller Regel langsamer und verbrauchen mehr Speicherplatz als Programme, die mit Schleifen, Variablen und Zustandsänderungen von Objekten arbeiten. Das ist nicht überraschend – das imperative Programmiermodell ist in erster Linie am Rechner und seinen Eigenschaften orientiert, das deklarative Modell am zu lösenden Problem.

Perlis' Bonmot zielt noch auf einen weiteren Schwachpunkt der deklarativen Programmierung ab, der ernst genommen werden muss: Die Kosten – vor allem die Laufzeit – deklarativ formulierter Programme sind weniger leicht abzuschätzen als die von imperativen Programmen. Eines der bekanntesten Beispiele für dieses Phänomen liefern die Fibonacci-Zahlen. Sie sind durch das Bildungsgesetz

$$f_0 = 0, f_1 = 1, f_n = f_{n-2} + f_{n-1} \quad (n > 1)$$

definiert. Das kann man in t.Zero so schreiben:

```
; n-te Fibonacci-Zahl; n = 0,1,2,...
(define (fibonacci n)
  (if (< n 2) n
      (+ (fibonacci (- n 2)) (fibonacci (- n 1))))))
```

Wenn man diese Funktion ausprobiert, stellt man schnell fest, dass sie schon für moderate Werte von n unerträglich langsam ist.

Im Tracing-Modus sieht man den Grund: `fibonacci` ruft sich immer wieder mit denselben Argumenten auf. Tatsächlich ist die Zahl $a_{n,k}$, die angibt, wie oft `(fibonacci k)` während der Auswertung von `(fibonacci n)` aufgerufen wird, selbst eine Fibonacci-Zahl. Für $2 \leq k \leq n$ gilt die Beziehung $a_{n,k} = f_{n-k+1}$. Beispielsweise würde die Eingabe `(fibonacci 100)` zu $f_{99} = 218922995834555169026$ Aufrufen von `(fibonacci 2)` führen.

Die Funktion `fibonacci` scheint ein gutes Beispiel für die Ineffizienz deklarativer Programme zu sein.

Einspruch! Wer sagt eigentlich, dass die Auswertung von `(fibonacci n)` so wie in dem sehr schlicht implementierten t.Zero-Interpreter verlaufen muss? Das deklarative Programm `fibonacci` besagt dies jedenfalls nicht. Es schreibt lediglich vor, dass `(fibonacci n)` und `(+ (fibonacci (- n 1)) (fibonacci (- n 2)))` für $n \geq 2$ denselben Wert haben.

Wäre der Interpreter in der Lage, einmal errechnete Werte von Funktionen zu speichern, so würde sich herausstellen, dass das Programm `fibonacci` durchaus brauchbar ist.

In Kapitel 4 werden wir sehen, wie man Funktionen schreibt, die ein solches „Gedächtnis“ haben. Im Moment ist nur wichtig zu wissen, dass die Ineffizienz der Funktion `fibonacci` keineswegs an einer prinzipiellen Unterlegenheit des deklarativen Programmierstils liegt, und dass sie durch eine geeignete Implementierung der Programmiersprache vermieden werden kann.

Anstatt einfach dem Interpreter die Schuld zu geben, kann man aber auch nach anderen Auswegen suchen. Eine Möglichkeit besteht darin, die Fibonacci-Zahlen intelligenter zu

beschreiben. Dazu betrachten wir die Zahlenfolge, die durch

$$f_0 = a, f_1 = b, f_n = f_{n-2} + f_{n-1} \quad (n > 1)$$

bestimmt ist. Für $a = 0, b = 1$ sind das die Fibonacci-Zahlen. Für $n > 1$ ist die n -te Zahl dieser Folge zugleich die $(n-1)$ -te Zahl derselben Folge, wenn man mit $f_0 = b, f_1 = a + b$ beginnt und dann ebenfalls mit $f_n = f_{n-2} + f_{n-1}$ fortfährt.

Diese Tatsache benutzen wir bei der Formulierung als t.Zero-Programm:

```
; n-te Fibonacci-Zahl bei Start mit a, b anstelle von 0, 1
(define (generalized-fibo a b n)
  (if (= n 0) a
      (if (= n 1) b
          (generalized-fibo b (+ a b) (- n 1))))))

; n-te Fibonacci-Zahl
(define (fibonacci n) (generalized-fibo 0 1 n))
```

Die Berechnung der Fibonacci-Zahlen ist damit blitzschnell, auch f_{1000} braucht nur Millisekunden. Außerdem bekommen wir als Bonus die nach É. Lucas (S. 24) benannten Lucas-Zahlen:

```
; n-te Lucas-Zahl
(define (lucas n)
  (generalized-fibo 2 1 n))
```

Für die Lucas-Zahlen gilt die Beziehung $L_n = f_{n-1} + f_{n+1}$ ($n > 0$). Man könnte sie deshalb auch so berechnen:

```
; Lucas-Zahlen, alternative Definition
(define (lucas n)
  (if (= n 0) 2
      (+ (fibonacci (- n 1)) (fibonacci (+ n 1)))))
```

So wie es in der klassischen Programmierung für ein und denselben Zweck gute und weniger gute Algorithmen gibt, wobei die Bedeutung von „gut“ gar nicht immer eindeutig festliegt, denn unterschiedliche Ziele können in Konflikt miteinander stehen, so muss man auch bei deklarativer Programmierung nach der „richtigen“ Beschreibung des gewünschten Results suchen. Das ist nicht immer einfach.

1.4 Syntax und Semantik

In diesem Abschnitt betrachten wir die lexikalischen, syntaktischen und semantischen Regeln der Sprache t.Zero etwas genauer.

1.4.1 Lexikalische Struktur

t.Zero-Programme sind auf den ersten Blick Texte, also Folgen einzelner Tastaturzeichen. Aus Sicht der Grammatik der Sprache bilden jeweils mehrere aufeinander folgende Tastaturzeichen ein *Token*. Die Unterteilung des Textes in Token ist Aufgabe des *Lexers*.

Es gibt in t.Zero drei Arten von Token, die der Lexer unterscheiden muss:

- Zahlen

- Namen
- Linke und rechte runde Klammern

Runde Klammern werden im Programmtext durch sich selbst dargestellt. Zahlen und Namen sind zwar aus Sicht der Grammatik nicht weiter zerlegbar, für den Lexer haben sie aber eine innere Struktur.

Zahlen sind Folgen von Dezimalziffern, die an einer Stelle einen Dezimalpunkt oder einen Bruchstrich enthalten dürfen. Direkt vor einer Zahl, ohne Leerzeichen dazwischen, darf ein Vorzeichen stehen.

Namen sind in *t.Zero* Namen von Operatoren, von benutzerdefinierten Funktionen oder von formalen Parametern von Funktionen. Die lexikalische Struktur von Namen ist ein bisschen gewöhnungsbedürftig: Ein Name darf aus Buchstaben, Zahlen und/oder Sonderzeichen bestehen. Nicht alle Sonderzeichen sind erlaubt (beispielsweise keine runden Klammern). Die erlaubten Zeichen findet man in der Klasse `Lexer` im Quelltext des Interpreters:

```
// Sonderzeichen, die in Bezeichnern verwendet werden dürfen:
private final static String IDENT_CHAR = ".$/%&=?~\*+~@#<>|:,‘-“;
```

Ein Name kann nicht mit einer Ziffer beginnen, auch ein Anfang mit den Zeichen `+` oder `-` mit nachfolgenden Ziffern ist nicht möglich, sonst würde der Lexer eine Zahl lesen.

Von der im Vergleich zu Java großen Freiheit der Namenswahl sollte man tunlichst nur sparsamen Gebrauch machen. Die folgende Definition wäre erlaubt, aber nicht klug:

```
-> (define (. .) (* . .))
function[.]
-> (. .5)
0.25
```

Hier wird der Name `.` (der Punkt ist in den *t.Sprachen* tatsächlich ein Name) als Funktionsname und gleichzeitig als Name des formalen Parameters der Funktion verwendet. Quizfrage: Wären auch `+` oder `*` als Parameternamen möglich?

1.4.2 Syntax

Die Syntax von *t.Zero* ist extrem einfach. Sie wird durch die folgende Grammatik beschrieben:

```
tZero      -> Expression
Expression -> Number | Ident | List
List       -> ( Expression* )
```

In Worten: Eine Eingabe für *t.Zero* ist ein Ausdruck. Ein Ausdruck ist eine Zahl, ein Name oder eine geklammerte Liste aus keinem, einem oder mehreren Ausdrücken.

Dieselbe Syntax werden wir (mit einer einzigen Erweiterung, dem Quote-Zeichen) auch für alle anderen Sprachen dieses Buchs benutzen. Syntax hat nur wenig mit den Ausdrucksmöglichkeiten, also der semantischen Substanz, einer Programmiersprache zu tun.

Betrachten wir die folgende, ziemlich beliebig herausgegriffene Java-Anweisung:

```
0  try {
1    line = source.readLine();
2    pos = 0;
3    sval = "";
```

```

4 } catch (IOException ioe) {
5     String msg = "I/O-error:␣" + ioe.getMessage();
6     throw new RuntimeException(msg);
7 }

```

Man könnte dieselbe Anweisung ohne großen Verlust an Lesbarkeit in der Syntax von t.Zero ausdrücken. Das Folgende ist kein t.Zero-Code; es könnte aber zum Beispiel in t.Scheme realisiert werden:

```

(try-catch
 (block
  (:= line (source readLine))
  (:= pos 0)
  (:= sval **))
 (IOException ioe)
 (block
  (String msg (+ "I/O-error: " (ioe getMessage)))
  (throw (new RuntimeException msg)))))

```

Das einzige Problem bei dieser Notation sind die Klammergebirge, vor allem am Ende von Ausdrücken. Wenn man einen guten Editor verwendet, der bei der Eingabe einer schließenden rechten Klammer die zugehörige öffnende linke Klammer markiert, verschwindet dieses Problem nach kurzer Eingewöhnungszeit weitgehend.

Zurück zur Grammatik von t.Zero. Ihre Token sind *Number*, *Ident*, linke und rechte Klammer. Die beiden Klammern stellen wir durch sich selbst dar; sie sind in der Grammatik unterstrichen, um sie von den runden Klammern zu unterscheiden, die in der formalen Beschreibung von Grammatiken als Meta-Symbole vorkommen dürfen.

Number und *Ident* haben als Token syntaktisch gesehen keine innere Struktur. Der Lexer gibt ihnen aber ihren jeweiligen speziellen Wert als *Attribut* mit, er wird für die Auswertung des Ausdrucks – also für seine Semantik – benötigt.

Die Prüfung, ob eine Folge von Token, die vom Lexer kommt, auch tatsächlich einen Ausdruck im Sinne dieser Grammatik darstellt, wird von einem Programmteil durchgeführt, den man als *Parser* bezeichnet.

Aus einer korrekten, dieser Grammatik entsprechenden Eingabe, die zunächst nur aus Tastaturzeichen besteht, erzeugt der Parser den entsprechenden Ausdruck als eigenständiges Objekt. Beispiel: Die Eingabe sei

(+ 12 3)

Daraus erzeugt der Lexer fünf Token (die Attribute stehen als Indizes daneben):

((S) + (N)₁₂ (N)₃)

Der Parser beginnt damit, die Regel für *Expression* zu parsen. Er sieht als Erstes das Token (((linke Klammer) und beginnt die Regel für *List* zu parsen. Er parst wieder die Regel für *Expression*, erkennt das Token (S) (*Ident*), parst noch zweimal *Expression* und erkennt dabei zweimal ein Token (N) (*Number*). Anhand des Tokens () stellt er schließlich fest, dass keine weiteren Ausdrücke in der Liste zu parsen sind.

Während des Parsens der Regel für *List* erzeugt der Parser zugleich auch die Liste als Kette von Objekten, die durch Referenzen verknüpft sind:



Dabei werden vom Parser Objekte der Typen `List`, `Ident` etc. erzeugt, welche die verschiedenen Arten von Eingabe-Ausdrücken von *t.Zero* repräsentieren. Bei der Erzeugung dieser Objekte gibt der Parser den Konstruktoren als Argumente die vom Lexer gelieferten Attribute mit.

Es sieht auf den ersten Blick nicht so aus, als sei aus der Eingabe `(+ 12 3)` bei der Umwandlung in eine Liste überhaupt etwas wirklich Neues entstanden. Tatsächlich wurde aber aus einer einfachen Zeichenkette ein Objekt mit einer detaillierten inneren Struktur gebildet. Es kann im nächsten Schritt, der *semantischen Phase*, ausgewertet werden.

1.4.3 Semantik

Die Datentypen einer Programmiersprache und die Operatoren, die für den Umgang mit ihnen zur Verfügung stehen, sind gewissermaßen das Rohmaterial beim Programmieren. Sie bestimmen in einem hohen Maße die Ausdrucksmöglichkeiten der Sprache.

Die wesentlichen Datentypen von *t.Zero* sind Zahlen und Wahrheitswerte¹. Die Operatoren von *t.Zero* haben wir alle schon benutzt:

1. Die numerischen Operatoren `+`, `-`, `*` und `/`. Sie erwarten jeweils zwei Zahlen bzw. Ausdrücke, deren Wert eine Zahl ist, als Argumente und liefern eine Zahl als Wert.
2. `(round x N)` wandelt eine Zahl `x` in eine `N`-stellige Dezimalzahl um. Dabei wird zur Null hin gerundet.
3. `(num x)` und `(den x)` für den Zugriff auf Zähler und Nenner von `x`.
4. Die Vergleichsoperatoren `=` und `<`. Sie erwarten zwei Zahlen als Argumente, der Rückgabewert ist vom Typ `Boolean`. Mit `=` darf man beliebige Ausdrücke miteinander vergleichen.
5. Der bedingte Ausdruck `(if bool expr alt-expr)`. Für `bool` muss ein Ausdruck stehen, dessen Wert vom Typ `Boolean` ist. Wenn dieser Wert `true` ist, so wird der Ausdruck `expr` ausgewertet und sein Wert als Resultat des bedingten Ausdrucks zurückgegeben. Andernfalls wird der alternative Ausdruck `alt-expr` ausgewertet und dessen Wert ist das Resultat.
6. `(define function-head function-body)`. Der Funktionskopf muss eine nichtleere Liste von Namen sein, der Rumpf darf ein beliebiger Ausdruck sein. Aus Kopf und Rumpf wird ein Objekt erzeugt, das die Funktion im Interpreter repräsentiert. Dieses Objekt wird in einer Tabelle gespeichert, wobei der Name der Funktion als Suchschlüssel zum Wiederfinden dient.

Eigentlich gehört auch der Operator `trace` in diese Aufzählung; er hat aber für die Semantik der Sprache keine Bedeutung.

Die Umsetzung der Semantikregeln im Verlauf der Auswertung eines Ausdrucks ist die Sache des Interpreters. Seine Arbeitsweise wird im nächsten Abschnitt diskutiert.

¹ Daneben gibt es noch Namen und Funktionen als weitere Datentypen. Wenn man den Namen einer Funktion eingibt, erhält man die Funktion selbst als Resultat – mit dem man aber nicht viel anfangen kann. Genau betrachtet haben auch Fehler einen eigenen Typ. Darauf kommen wir später zurück.



1.5 Der Interpreter

1.5.1 Die Read-Eval-Print-Schleife

Der Interpreter ist die zentrale Instanz bei der Arbeit mit t.Zero. Beim Aufruf des Programms wird ein Objekt der Klasse `Interpreter` erzeugt. Es enthält als wichtigste Bestandteile den `Lexer`, den `Parser` und die globale Umgebung.

Damit sind vier der für die Implementierung wichtigsten Klassen benannt:

1. Die Klasse `Interpreter`. Sie enthält insbesondere die Methode `main`, die beim Aufruf des Programms `tzero` ausgeführt wird.
2. Die Klassen `Lexer` und `Parser`. Ihre Funktion wurde im vorigen Abschnitt beschrieben.
3. Die Klasse `Env` (für Environment). Eine Umgebung ist eine Tabelle, in der während der Ausführung Ausdrücke abgespeichert werden, und zwar so, dass man sie unter einem Namen wiederfinden kann. Im Wesentlichen handelt es sich um eine Hashtabelle mit Symbolen als Schlüsseln und Ausdrücken als Werten. In der globalen Umgebung werden zu Beginn die vordefinierten Operatoren der Sprache gespeichert, im weiteren Verlauf dann auch die benutzerdefinierten Funktionen.

Beim Programmstart wird also ein `Interpreter`-Objekt mit einer globalen Umgebung und einem `Parser` erzeugt (dieser enthält den `Lexer`). An diesen `Interpreter` wird sofort anschließend ein Methodenaufruf geschickt, der seine *Read-Eval-Print-Schleife* in Gang setzt.

Diese Schleife durchläuft bis zur Beendigung des Interpreters die folgenden drei Schritte:

1. Schreiben der Eingabeaufforderung und **Einlesen** einer Eingabe. Die Eingabe wird vom `Parser` in ein internes Objekt umgewandelt, das einen t.Zero-Ausdruck repräsentiert.
2. **Auswerten** des Ausdrucks mit Hilfe der in der globalen Umgebung gespeicherten Operatoren und Funktionen.
3. **Schreiben** des Resultats.

Während eines Schleifendurchlaufs kann es auf viele Arten zu Situationen kommen, die die reguläre weitere Ausführung unmöglich machen:

Es kann Fehler beim Einlesen geben, wenn beispielsweise die Eingabe illegale Zeichen enthält oder ein Lesefehler auftritt. Der `Parser` muss möglicherweise abbrechen, weil die Eingabe kein t.Zero-Ausdruck ist. Die meisten Fehler treten allerdings während der Auswertung auf. Sogar die Auswertung einer völlig korrekten Eingabe kann scheitern, zum Beispiel an Speicherplatzmangel. Fehler werden vom `Interpreter` durch eine entsprechende Exception-Verarbeitung abgefangen.

Man kann auch die Beendigung der Read-Eval-Print-Schleife, die ja eigentlich eine Endlosschleife ist, als einen „Fehler“ auffassen, bei dem eine Exception erzeugt wird. Beim Auftreten dieser Exception wird die Schleife verlassen.

Insgesamt kann man sich die Implementierung der Read-Eval-Print-Schleife in Java etwa in der folgenden Form vorstellen:

```

0 // Read-Eval-Print-Schleife
1 void run() {
2     while (true) {
3         try {
4             writePrompt();
5             Expr expr = parser.read();
6             Expr result = expr.eval(globalEnv);

```

```

7         print(result);
8     } catch (RecoverableException re) {
9         ... // Behandlung einer 'normalen' Fehlersituation
10    } catch (QuitException qe) {
11        break; // Read-Eval-Print-Schleife verlassen
12    }
13 }
14 }

```

Diese Formulierung sagt nichts darüber aus, *wie* der Ausdruck *expr* ausgewertet wird. Man sieht aber zumindest, dass zu seiner Auswertung eine Umgebung notwendig ist. In der Read-Eval-Print-Schleife ist das die globale Umgebung. Wir werden gleich sehen, dass an der Auswertung von Ausdrücken auch noch andere Umgebungen beteiligt sein können.

1.5.2 Ausdrücke und ihre Auswertung

Die Auswertung von Zahlen und Namen

In *t.Zero* werden drei Arten von Ausdrücken ausgewertet: Zahlen, Namen und Listen. Die Auswertung von Zahlen und Namen ist einfach:

Jede Zahl ist ihr eigener Wert. Zahlen sind *Literale*, sie stehen für sich selbst. In der Java-Klasse für *t.Zero*-Zahlen sieht das so aus:

```

0 public Expr eval(Env env) {
1     return this;
2 }

```

Namen werden anhand der jeweiligen Umgebung ausgewertet: Ist bei einem Aufruf *expr.eval(env)* der Ausdruck *expr* ein Name, so wird in der Umgebung *env* nachgesehen, ob zu diesem Namen ein Wert gespeichert ist. Dieser Wert ist der Wert des Namens.

Der entsprechende Java-Code in der Klasse für *t.Zero*-Namen ist wieder sehr einfach:

```

0 public Expr eval(Env env) throws Alarm {
1     return env.retrieve(this);
2 }

```

Ein Alarm wird bei dieser Auswertung dann ausgelöst, wenn in der Umgebung *env* kein Wert zum Namen *this* gespeichert ist. Die Klasse *Alarm* ist von *Exception* abgeleitet. Sie repräsentiert in der Java-Implementierung die Ausnahmesituationen, die im Interpreter auftreten können.

Beispiel:

```

-> +
op[plus]
-> !=
Error: Unbound symbol !=

```

Der Name *+* ist auswertbar, er hat als Wert den Additionsoperator. Der Name *!=* ist in *t.Zero* nicht bekannt. Man könnte ihn als Name einer Funktion verwenden und ihn damit dem Interpreter mitteilen, z. B. so:

```

-> (define (!= x y) (not (= x y)))
function[!=]
-> !=
function[!=]

```

Der wesentliche Effekt einer Funktionsdefinition besteht darin, den Namen der Funktion an die Funktion selbst (das interne Objekt, das sie repräsentiert) zu binden. Anders gesagt: In der globalen Umgebung wird der Funktionsname als Suchschlüssel für die Funktion eingetragen.

Die Auswertung von Listen

Die Auswertung von Listen ist nicht ganz so einfach. Zwei Fälle sind zu unterscheiden:

- die Definition von Funktionen und
- die Anwendung von Funktionen und Operatoren.

Betrachten wir als Beispiel zwei Definitionen mit anschließender Anwendung einer Funktion:

```
-> (define (square x) (* x x))
function[square]
-> (define (square-sum a b) (+ (square a) (square b)))
function[square-sum]
-> (square-sum (+ 1 2) (* 2 2))
25
```

Die Auswertung der letzten Eingabe findet in einer Umgebung statt, in der die Namen `+`, `*`, `square` und `square-sum` definiert sind. Im Detail passiert das Folgende:

1. Der Kopf von `(square-sum (+ 1 2) (* 2 2))` wird ausgewertet, der Wert ist die Funktion `function[square-sum]`. Hier muss man gut unterscheiden zwischen dem Namen der Funktion und der Funktion selbst, die ein Objekt im Interpreter ist. Diese Funktion wird auf ihre Argumente *angewendet*.
2. Bei dieser Anwendung werden als Erstes die Argumente ausgewertet, in diesem Fall die beiden Ausdrücke `(+ 1 2)` und `(* 2 2)`. Deren Auswertung verläuft nach demselben Schema, sie ergibt die Werte 3 und 4.
3. Nun wird die globale Umgebung um zwei Einträge erweitert: die Symbole `a` mit dem Wert 3 und `b` mit dem Wert 4. Mit dieser erweiterten Umgebung wird nun der Rumpf der Funktion `square-sum`, die Liste `(+ (square a) (square b))`, ausgewertet.
4. Das führt zur Auswertung des Symbols `+` und der Ausdrücke `(square a)` und `(square b)`. Dabei wiederholt sich das Spiel: Bei der Auswertung von `(square a)` wird eine Umgebung erzeugt, in welcher der Parameter `x` der Funktion `square` den Wert 3 hat. Bezüglich dieser Umgebung wird der Rumpf von `function[square]`, also die Liste `(* x x)`, zu 9 ausgewertet. Entsprechend wird `(square b)` zu 16 ausgewertet. Auf diese Argumente wird schließlich `op[plus]` angewendet, das ergibt das Resultat 25.

Im Tracing-Modus kann man auch die Aufrufe der numerischen Operatoren verfolgen:

```
-> (trace + * square square-sum)
Tracing mode for op[plus] is on
Tracing mode for op[mult] is on
Tracing mode for function[square] is on
Tracing mode for function[square-sum] is on
-> (square-sum (+ 1 2) (+ 2 2))
  Call    (+ 1 2)
  Return  3 from (+ 1 2)
  Call    (+ 2 2)
  Return  4 from (+ 2 2)
```

```

Call    (square-sum 3 4)
Call    (square 3)
Call    (* 3 3)
Return  9 from (* 3 3)
Return  9 from (square 3)
Call    (square 4)
Call    (* 4 4)
Return  16 from (* 4 4)
Return  16 from (square 4)
Call    (+ 9 16)
Return  25 from (+ 9 16)
Return  25 from (square-sum 3 4)

```

25

Wie man sieht, wird ein Funktionsaufruf im Tracing-Modus erst nach der Auswertung aller Argumente der Funktion angezeigt. Der Aufruf beginnt aber schon in dem Moment, wenn der Kopf der Liste ausgewertet wird. Die Zeile `Call . . .` wird erst zu dem Zeitpunkt ausgegeben, in dem die Auswertung des Rumpfs der Funktion beginnt – also nach der Auswertung der Parameter.

Diese extrem detaillierte Beschreibung der Auswertung einer Liste suggeriert eine Komplexität, die sich in Wirklichkeit nur durch die mehrfach ineinandergeschachtelten Auswertungen ergibt.

Wir fassen deshalb nochmals zusammen. Eine Liste wird in drei Schritten ausgewertet:

1. Es wird kontrolliert, dass die Liste nicht leer ist.
2. Der Kopf der Liste wird ausgewertet, der Wert muss ein Operator oder eine Funktion sein. Operatoren und Funktionen werden auch als *Prozeduren* bezeichnet.
3. Die Prozedur wird auf die Argumente *angewendet*. Das Resultat der Anwendung ist der Wert der Liste.

Der Java-Code dazu könnte ungefähr so aussehen:

```

0 public Expr eval(Env env) throws Alarm {
1     checkEvalnil(this);
2     Expr expr = head.eval(env);
   Liste.
3     Procedure proc = checkProcedure(expr);
4     return proc.apply(tail, env);
   Liste.
5 }

```

Anwenden – hier implementiert durch die Methode `apply` – hat je nach Art der Prozedur eine unterschiedliche Bedeutung:

Bei einer Funktion werden die Argumente ausgewertet, aus den formalen Parametern und den Argumentwerten wird eine neue Umgebung gebildet und mit dieser Umgebung der Rumpf der Funktion ausgewertet.

Die Operatoren für Zahlen, also `op[plus]`, `op[mult]` etc., verhalten sich wie benutzerdefinierte Funktionen. Alle Argumente werden ausgewertet.

Anders verläuft die Anwendung bei den Operatoren `if` und `define`. Die Details hatten wir schon auf S. 38 besprochen: Bei `if` wird das erste Argument ausgewertet und in Abhängigkeit vom booleschen Wert anschließend entweder das zweite oder das dritte Argument. Bei `define` bleiben beide Argumente unausgewertet. Aus ihnen wird eine Funktion erzeugt, die in der globalen Umgebung gespeichert wird. Die neue Funktion ist zugleich das Resultat.

Aktive oder passive Auswertung?

Eine Überlegung soll noch erwähnt werden, die beim Entwurf der Implementierung des Interpreters eine Rolle gespielt hat. Sie hat nichts zu tun mit dem Funktionieren aus der Sicht der Benutzer, sondern betrifft ein ausschließlich internes, aber trotzdem wichtiges Detail.

Wir hatten in den kurzen Ausschnitten aus dem Java-Code des Interpreters gesehen, dass jede Java-Klasse, die einen Eingabetyp der Sprache (Zahlen, Namen und Listen) repräsentiert, eine eigene, für diesen Typ spezifische Methode `eval` zur Auswertung besitzt.

Man könnte den Interpreter ganz anders entwerfen: So wie es für die lexikalische Phase einen Lexer und für die Syntaxanalyse einen Parser gibt, könnte man für die Auswertungsphase einen Evaluator vorsehen.

Beim Start würde dann neben dem Lexer und dem Parser noch ein Evaluator erzeugt:

```
0 Evaluator evaluator = new Evaluator();
```

Dem Evaluator würden die vom Parser erzeugten Ausdrücke jeweils zur Auswertung überreicht. Die Ausdrücke würden zu ihrer Auswertung gewissermaßen nicht mehr selbst aktiv, sondern sie würden passiv ausgewertet.

Der Code der Read-Eval-Print-Schleife (S. 39) würde sich auf den ersten Blick nur unwesentlich verändern. Anstelle von

```
0 Expr result = expr.eval(globalEnv);
```

würde folgende Zeile stehen:

```
0 Expr result = evaluator.eval(expr, globalEnv);
```

Das macht den Eindruck eines unwichtigen Details. Tatsächlich hätte eine solche Änderung aber erhebliche Folgen für die Implementierung der t.Sprachen: Die Klasse `Evaluator`, die den Evaluator implementiert, müsste für jede der etwa 20 Arten von Ausdrücken, die es in den t.Sprachen gibt, wissen, wie deren Auswertung vorzunehmen ist. Das entspricht nicht dem Prinzip, Klassen zu entkoppeln, sie also von möglichst wenigen anderen Klassen abhängig zu machen.

Aktive Ausdrücke, bei denen die Art ihrer Auswertung in der jeweiligen Klasse spezifiziert ist, entsprechen viel besser den Prinzipien der Kapselung und Wiederverwendbarkeit, die die Grundlage der objektorientierten Programmierung bilden. Deshalb wurde für die Java-Implementierung der t.Sprachen das Prinzip der aktiven Auswertung gewählt.

Programmiersprachen – Konzepte, Strukturen und
Implementierung in Java

Clausing, A.

2011, XII, 450 S. 40 Abb., Softcover

ISBN: 978-3-8274-2850-9