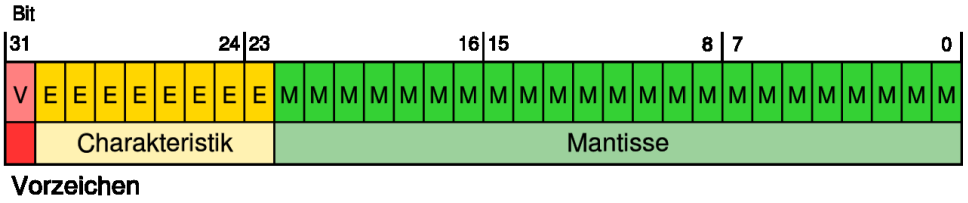
Informationen zur Übung 1

Rationale Zahlen nach IEEE 754

Die Norm IEEE 754[1] legt für eine Wortgröße von 32 Bit die folgende Form für Gleitkommazahlen fest:



Die Anzahl der Bits des Exponenten legt die minimalen und maximalen Werte der repräsentierten Zahlen fest, wohingegen die Anzahl der Bits der Mantisse die relative Genauigkeit der Werte bestimmt. Für unterschiedliche Größen der Exponenten und Mantissen legt die Norm folgendes fest:

Größe $(1+r+p)$	Exponent (r)	Mantisse (p)	Werte des Exponenten	größte Zahl
16 bit	5 bit	10 bit	$-14 \leq e \leq 15$	—
32 bit	8 bit	23 bit	$-126 \leq e \leq 127$	$\approx 2^{128} \approx 3,4 \cdot 10^{38}$
64 bit	11 bit	52 bit	$-1022 \leq e \leq 1023$	$\approx 2^{1024} \approx 1,8 \cdot 10^{308}$
128 bit	15 bit	112 bit	$-16382 \leq e \leq 16383$	—

Nach IEEE 754 sind die Zahlen normalisiert. wikipedia: Die Mantisse besteht aus den ersten n wesentlichen Ziffern der Binärdarstellung der noch nicht normalisierten Zahl. Die erste wesentliche Ziffer ist die höchstwertige (d.h. am weitesten links stehende) Ziffer, die von 0 verschieden ist. Da eine von 0 verschiedene Ziffer im Binärsystem nur eine 1 sein kann, muss diese erste 1 nicht explizit abgespeichert werden; gemäß der Norm IEEE 754 werden nur die folgenden p Ziffern gespeichert, die erste Ziffer ist eine implizite Ziffer oder ein implizites Bit (engl. hidden bit). Dadurch wird gewissermaßen 1 Bit Speicherplatz „gespart“. Die Interpretation einer Zahl ist

$$\pm(1, \text{Ziffern der Mantisse}) \cdot 2^{\text{Wert des Exponenten}}.$$

Der Wert $3,25 = (11,01)_2$ wird also als $(1,101)_2 \cdot 2^1$ gespeichert, wobei die Ziffer vor dem Komma nicht gespeichert wird, es ist das Hidden-Bit.

Beim Rechnen mit Gleitkommazahlen treten zwei unterschiedliche Arten von Ungenauigkeiten auf. Die erste ergibt sich durch die Umwandlung von der dezimalen in die binäre Darstellung, die zweite durch Denormalisierung und Rundung bei arithmetischen Operationen.

Umwandlung dezimal nach binär Der Wert $0,1$ dezimal kann nicht exakt dargestellt werden als Binärzahl. Es gilt $0,1 = (0,0\overline{0011})_2$, daher muss der Wert bspw. bei 32-bit `float` nach 23 Stellen abgeschnitten oder gerundet werden. Diese Form der Ungenauigkeiten lässt sich nicht vermeiden. Der folgende Programmabschnitt

[1] `https://de.wikipedia.org/wiki/IEEE_754`

```
float f = 0.1F;
printf(" f = %56.40f\n", f);
double d = 0.1;
printf(" d = %56.40lf\n", d);
```

liefert auf unserem Rechner die folgenden Werte:

```
f =          0.1000000014901161193847656250000000000000
d =          0.1000000000000000005551115123125782702118 2
```

Der binär gespeicherte Wert ist $(1,10011001100110011001101)_2 \cdot 2^{-4}$ in normalisierter Darstellung. Das erste Bit vor dem Komma ist das Hidden-Bit. Die hinteren, niederwertigen Stellen von $0,1 = (0,0\overline{0011})_2$ wurden nicht einfach abgeschnitten, denn dann hätte sich ein Wert kleiner als $0,1$ ergeben, sondern es wurde gerundet.

Rundung Bei binären Rundungen muss laut IEEE 754 zur nächstgelegenen darstellbaren Zahl gerundet werden. Wenn diese nicht eindeutig definiert ist (genau in der Mitte zwischen zwei darstellbaren Zahlen), wird so gerundet, dass das niederwertigste Bit der Mantisse 0 wird. Statistisch wird dabei in 50% der Fälle auf-, in den anderen 50% der Fälle abgerundet. (laut wikipedia)

Die obige Ausgabe des C-Programms kann je nach Hardware, Betriebssystem und Compiler variieren, da C keine exakten Vorgaben für die Datentypen macht, sondern nur Mindestanforderungen definiert.

Arithmetische Operationen Eine weitere Ungenauigkeit tritt bei der Addition von Gleitkommazahlen auf, und zwar durch die Denormalisierung der kleineren Zahl. Das Rechnen mit solchen Zahlen wird auch in dem Buch *Eine Einführung in die Mathematik an Beispielen aus der Informatik* von Goebbels und Rethmann beschrieben. Hier ein kurzer Auszug daraus:

Jetzt rechnen wir mit Gleitpunktzahlen. Schauen wir uns zunächst die Multiplikation und Division für Zahlen $x = m_x \cdot 2^{d_x}$ und $y = m_y \cdot 2^{d_y}$ an. Aus den Rechenregeln für Potenzen reeller Zahlen ergibt sich:

- Bei der Multiplikation werden die Mantissen multipliziert und die Exponenten addiert.

$$x \cdot y = (m_x \cdot m_y) \cdot 2^{d_x + d_y}$$

- Bei der Division werden die Mantissen dividiert und die Exponenten subtrahiert.

$$x : y = (m_x : m_y) \cdot 2^{d_x - d_y}$$

Betrachten wir auch dazu wieder ein Beispiel, zunächst für die Multiplikation $121,8 \cdot 0,37$ im Dezimalsystem, weil uns das vertrauter ist:

$$(1,218 \cdot 10^2) \ \cdot \ (3,7 \cdot 10^{-1}) = 4,5066 \cdot 10^1 = 45,066$$

Und nun ein Beispiel für die Division $450,66 : 37$ im Dezimalsystem:

$$(4,5066 \cdot 10^2) \ : \ (3,7 \cdot 10^1) = 1,218 \cdot 10^1 = 12,18.$$

Die Multiplikation und Division der Zahlen erfordert keine angepasste Darstellung. Das ist bei der Addition und Subtraktion der Zahlen $x = m_x \cdot 2^{d_x}$ und $y = m_y \cdot 2^{d_y}$ anders. Wenn wir schriftlich addieren und subtrahieren, dann muss das Komma bzw. der Dezimalpunkt an der gleichen Stelle stehen. Bei Gleitpunktzahlen wird die Position des Kommas durch den Exponenten bestimmt, so dass wir diesen angleichen müssen. Rechnerisch sieht das dann so aus:

$$x \pm y = \begin{cases} (m_x \cdot 2^{d_x - d_y} \pm m_y) \cdot 2^{d_y}, & \text{falls } d_x \leq d_y \\ (m_x \pm m_y \cdot 2^{d_y - d_x}) \cdot 2^{d_x}, & \text{falls } d_x > d_y. \end{cases}$$

Wie man hier sieht, wird der kleine Exponent dem großen Exponenten angeglichen. Es macht keinen Sinn, den großen Exponenten an den kleinen anzupassen, da dadurch eventuell die höchstwertigen Stellen nicht mehr dargestellt werden könnten. Man nimmt lieber in Kauf, niederwertigere Stellen zu verlieren, und akzeptiert damit Rundungsungenauigkeiten. Bei einer Mantisse von 4 Stellen (ohne Vorzeichen) zur Basis 10 seien x, y, z wie folgt gegeben:

$$x := +1,235 \cdot 10^2, \quad y := +5,512 \cdot 10^4, \quad z := -5,511 \cdot 10^4.$$

Dann erhalten wir mit der Rundungsungenauigkeit $1,235 \cdot 10^{-2} \approx 0,012$:

$$\begin{aligned} x + y &= +1,235 \cdot 10^2 + 5,512 \cdot 10^4 = (+1,235 \cdot 10^{-2} + 5,512) \cdot 10^4 \\ &\approx (+0,012 + 5,512) \cdot 10^4 = +5,524 \cdot 10^4, \end{aligned}$$

$$(x + y) + z \approx +5,524 \cdot 10^4 - 5,511 \cdot 10^4 = +0,013 \cdot 10^4 = +1,300 \cdot 10^2.$$

Andererseits tritt bei der folgenden Rechnung kein Rundungsfehler auf:

$$y + z = +5,512 \cdot 10^4 - 5,511 \cdot 10^4 = +0,001 \cdot 10^4 = +1,000 \cdot 10^1,$$

$$\begin{aligned} x + (y + z) &= +1,235 \cdot 10^2 + 1,000 \cdot 10^1 = +1,235 \cdot 10^2 + 0,100 \cdot 10^2 \\ &= +1,335 \cdot 10^2 \neq +1,300 \cdot 10^2 = (x + y) + z. \end{aligned}$$

Das Assoziativgesetz gilt also nicht bei Gleitpunktzahlen mit endlicher Stellenzahl. Es ist sinnvoll, Zahlen mit ähnlich großen Exponenten zu addieren oder subtrahieren.

Ein anderes Beispiel, bei dem die beschränkte Stellenzahl der Mantisse Auswirkungen hat, erhalten wir bei Berechnung der linken Seite von

$$\frac{(x+y)^2 - x^2 - 2xy}{y^2} = \frac{x^2 + 2xy + y^2 - x^2 - 2xy}{y^2} = 1$$

auf einem 32 Bit Computer mit einfacher Genauigkeit (`float`) für die Zahlen $x = 1000$ und $y = 0,03125 = \frac{1}{32}$. Obwohl beide Zahlen exakt dargestellt werden können, bekommen wir das Ergebnis 0. Das liegt daran, dass $(x+y)^2$ eine große Zahl und y^2 eine im Vergleich so kleine Zahl ist, dass in der Zahlendarstellung aufgrund der Stellenzahl der Mantisse genau der y^2-Anteil von $(x+y)^2 = x^2 + 2xy + y^2$ nicht mehr gespeichert werden kann. Daher entsteht bereits im Zähler die Null. Um solche groben Fehler zu vermeiden, sollten Zahlen bei Addition und Subtraktion ungefähr die gleiche Größenordnung haben.

Wir können festhalten, dass Ungenauigkeiten im Umgang mit Gleitpunktzahlen sowohl bei der Umwandlung vom Dezimal- ins Dualsystem als auch bei den arithmetischen Operationen auftreten. (Ende des Auszugs aus dem oben genannten Buch.)

Schauen wir uns einmal an, was bei einer fortgesetzten Multiplikation mit 10 passiert. Dazu schreiben wir zunächst ein kleines Programm:

```
float f = 0.1F;
for (int i = 0; i < 10; i++) {
    printf(" f = %56.40f\n", f);
    f *= 10;
}
```

Betrachten wir nun einen Teil der Ausgabe des Programms.

```
f =                0.1000000014901161193847656250000000000000
f =                1.0000000000000000000000000000000000000000
f =               10.0000000000000000000000000000000000000000
f =              100.0000000000000000000000000000000000000000
f =             1000.0000000000000000000000000000000000000000
f =            10000.0000000000000000000000000000000000000000
```

Da staunt der Fachmann und der Laie wundert sich: Zunächst starten wir mit einem Wert, der nur annähernd dem Wert $0,1$ entspricht. Das hatten wir uns oben schon überlegt, soweit ist alles wie erwartet. Aber das Ergebnis der ersten Multiplikation überrascht schon sehr, denn es wird nicht

$$10 \cdot 0,1000000014901161193847656250 = 1,0000000014901161193847656250$$

berechnet, stattdessen wird $10 \cdot 0,1 = 1,0$ berechnet. Alle weiteren Multiplikationen sind dann korrekt. Wie kann das sein? Schauen wir uns dazu an, wie eine Multiplikation[2] mit $10 = (1010)_2$ im Rechner abläuft.

	1,	1001	1001	1001	1001	1001	101	$[\cdot 2^{-4}]$	$\cdot(1010)_2$
=	1,	1001	1001	1001	1001	1001	101	$[\cdot 2^{-3}]$	
+	1,	1001	1001	1001	1001	1001	101	$[\cdot 2^{-1}]$	
=	0,	0110	0110	0110	0110	0110	01101	$[\cdot 2^{-1}]$	(Denormalisieren)
+	1,	1001	1001	1001	1001	1001	101	$[\cdot 2^{-1}]$	
=	10,	0000	0000	0000	0000	0000	00001	$[\cdot 2^{-1}]$	
=	1,	0000	0000	0000	0000	0000	000001	$[\cdot 2^{0}]$	(Normalisieren)
=	1,	0000	0000	0000	0000	0000	000001	$[\cdot 2^{0}]$	(Runden)
=	1,	0000	0000	0000	0000	0000	000	$[\cdot 2^{0}]$	

Das rot markierte Bit ist die 23. Stelle der Mantisse. Diese ist für das Runden wichtig. Wenn auf dieses Bit, also das niederwertigste Bit der Mantisse, eine 0 folgt, dann ist die Zahl näher an der betragsmäßig kleineren darstellbaren Zahl. Wenn auf das niederwertigste Bit eine 1 folgt und an irgendeiner Stelle dahinter noch eine weitere 1 steht, dann ist die Zahl näher an der betragsmäßig größeren Zahl. Wenn das Bit hinter dem niederwertigsten Bit eine 1 ist und an keiner weiteren Stelle dahinter noch eine 1 steht, dann ist die Zahl genau zwischen 2 darstellbaren Zahlen. Dann wird so gerundet, dass nach dem Runden eine 0 im niederwertigsten Bit steht.

Wenn wir den Wert $0,1$ in einen Bruch umwandeln wollen, dann nutzen wir das in der Übungsaufgabe beschriebene Verfahren: Den Bruch fortlaufend Erweitern, bis im Zähler keine Nachkommastelle mehr enthalten ist, anschließend wird der Bruch gekürzt. In diesem Fall erhalten wir:

$$0,1 = \frac{0,1}{1} = \frac{1}{10}$$

[2] Im Internet kann das bspw. unter `https://zahlensysteme-rechner.de/ieee-754-addieren/` visualisiert werden.

Listing 1: Umwandeln eines `float`-Wertes in einen Bruch.

```
1  float valueOf(bruch_t *r, float z) {
2      int n = 1;
3
4      printf("%40.25lf\n", z);
5      while (z - trunc(z) != 0) {
6          z *= 10;
7          n *= 10;
8          printf("%40.25lf\n", z);
9      }
10     r->_z = z;
11     r->_n = n;
12     reduce(r);
13     return z;
14 }
```

Bei Ausführung der Funktion `valueOf` aus Listing 1, erhalten wir die folgende Ausgabe:

```
                  0.1000000014901161193847656
                  1.0000000000000000000000000
    0.1 = 1 / 10
```

Alles ist so, wie gewünscht. Durch das Wegfallen der ungewünschten Nachkommastellen haben wir einen exakten Bruch erhalten, obwohl zunächst der Wert $0,1$ nicht exakt gespeichert werden konnte.

Schauen wir uns nun ein anderes Beispiel an. Wir wollen den Wert $3,0078125$ in einen Bruch umwandeln. Nach dem in der Übungsaufgabe beschriebenen Verfahren des iterativen Erweiterns und dem anschließenden Kürzen des Bruchs, hier $ggT(30.078.125, 10.000.000) = 78.125$, erhalten wir bei exakter Rechnung:

$$\begin{aligned} 3,0078125 &= \frac{30,078125}{10} = \frac{300,78125}{100} = \frac{3007,8125}{1000} = \frac{30078,125}{10000} \\ &= \frac{300781,25}{100000} = \frac{3007812,5}{1000000} = \frac{30078125}{10000000} = \frac{385}{128} \end{aligned}$$

Bei Ausführung des obigen Programms erhalten wir aber die folgende Ausgabe:

```
                  3.0078125000000000000000000
                 30.0781250000000000000000000
                300.7812500000000000000000000
               3007.8125000000000000000000000
              30078.1250000000000000000000000
             300781.2500000000000000000000000
            3007812.5000000000000000000000000
           30078124.0000000000000000000000000
```

In der letzten Zeile ergibt sich eine Ungenauigkeit, sodass wir den folgenden, gekürzten Bruch erhalten:

$$3,0078125 \approx \frac{7519531}{2500000}$$

Schauen wir uns nun an, wie die Ungenauigkeit bei der letzten Multiplikation mit $10 = (1010)_2$ zustande kommt. Es gilt $3007812,5 = (1,01101111001010100010010)_2 \cdot 2^{21}$. Das kann bspw. unter `https://www.matheretter.de/rechner/dezimalbinar` geprüft werden.

	1,	0110	1111	0010	1010	0010	010	$[\cdot 2^{21}]$ $\cdot(1010)_2$
=	1,	0110	1111	0010	1010	0010	010	$[\cdot 2^{22}]$
+	1,	0110	1111	0010	1010	0010	010	$[\cdot 2^{24}]$
=	0,	0101	1011	1100	1010	1000	10010	$[\cdot 2^{24}]$ (Denormalisieren)
+	1,	0110	1111	0010	1010	0010	010	$[\cdot 2^{24}]$
=	1,	1100	1010	1111	0100	1010	11010	$[\cdot 2^{24}]$ (Abrunden)
=	1,	1100	1010	1111	0100	1010	110	$[\cdot 2^{24}]$
=	30.078.124							

Problem hier ist, dass die Zahlen zu groß werden und mit einer 23-bit großen Mantisse nicht dargestellt werden können. Die Mantisse besteht nur aus 23 Bit, aber die Mantisse wird mit 2^{24} multipliziert, das Komma also um 24 Stellen nach rechts verschoben. Ungerade Zahlen können so nicht mehr dargestellt werden, es fehlt die entsprechende Stelle in der Mantisse. Es entstehen also „Lücken“ im Zahlenbereich. Man sieht das auch an dem folgenden Programm:

```c
#include <stdio.h>

int main(void) {
    int x = 16777212;
    float f = 16777212.0;

    printf("       x     |        f\n");
    printf("-------------+-------------\n");
    for (int i = 0; i < 10; i++) {
        printf(" %9d | %12.f\n", x, f);
        x += 1;
        f += 1.0f;
    }
    return 0;
}
```

Hier die Ausgabe des Programms:

```
       x     |        f
-------------+-------------
  16777212 |   16777212.0
  16777213 |   16777213.0
  16777214 |   16777214.0
  16777215 |   16777215.0
  16777216 |   16777216.0
  16777217 |   16777216.0
  16777218 |   16777216.0
  16777219 |   16777216.0
  16777220 |   16777216.0
  16777221 |   16777216.0
```

Gleitpunktzahlen vom Typ `float` haben Einbußen hinsichtlich der Genauigkeit im Vergleich zu ganzen Zahlen vom Typ `int`, da die Zahlenbereiche unterschiedlich groß sind, aber die selbe Anzahl Bit zur Codierung genutzt werden:

- größte Gleitpunktzahl bei 32 Bit: etwa 2^{127}
- größte Zahl in Dualdarstellung bei 32 Bit: etwa 2^{31}

Was das für die Programmierung heißt, sehen wir an dem obigen Programm. Einige Zahlen im Bereich von 0 bis 2^{31} können mittels `float` nicht dargestellt werden. Da die Mantisse aus nur 23 Bit besteht, stehen inklusive des Hidden-Bit nur 24 Bit zur Verfügung. Daher können nur die Werte von 0 bis 2^{24} dargestellt werden, danach hat die Darstellung „Lücken“, da die Nachkommastellen in der normalisierten Darstellung abgeschnitten werden.

Um den Problemen, die bei der Multiplikation mit 10 entstehen, aus dem Weg zu gehen, können Zähler und Nenner des Bruchs jeweils mit 2 multipliziert werden. Bei der iterativen Multiplikation mit 2 treten keine Ungenauigkeiten durch Denormalisierung auf, da die Multiplikation ohne zusätzliche Addition durchgeführt werden kann. Es wird einfach nur das Komma um eine Position verschoben, indem der Exponent um eins erhöht wird. Ändern wir also unser Programm wie folgt:

```
float valueOf(bruch_t *r, float z) {
    int n = 1;

    printf("%40.25lf\n", z);
    while (z - trunc(z) != 0) {
        z *= 2;  // !!!!!!!!!!
        n *= 2;  // !!!!!!!!!!
        printf("%40.25lf\n", z);
    }
    r->_z = z;
    r->_n = n;
    reduce(r);
    return z;
}
```

Betrachten wir die Ausgabe des Programms für den Wert $3,0078125$:

```
                3.0078125000000000000000000
                6.0156250000000000000000000
               12.0312500000000000000000000
               24.0625000000000000000000000
               48.1250000000000000000000000
               96.2500000000000000000000000
              192.5000000000000000000000000
              385.0000000000000000000000000
  3.0078125 = 385 / 128
```

Oder als Binärzahlen:

$$
\begin{aligned}
& 1,10000001000000000000000 && [\cdot 2^{1}] \\
\cdot 2 \hookrightarrow\ & 1,10000001000000000000000 && [\cdot 2^{2}] \\
\cdot 2 \hookrightarrow\ & 1,10000001000000000000000 && [\cdot 2^{3}] \\
\cdot 2 \hookrightarrow\ & 1,10000001000000000000000 && [\cdot 2^{4}] \\
\cdot 2 \hookrightarrow\ & 1,10000001000000000000000 && [\cdot 2^{5}] \\
\cdot 2 \hookrightarrow\ & 1,10000001000000000000000 && [\cdot 2^{6}] \\
\cdot 2 \hookrightarrow\ & 1,10000001000000000000000 && [\cdot 2^{7}] \\
\cdot 2 \hookrightarrow\ & 1,10000001000000000000000 && [\cdot 2^{8}]
\end{aligned}
$$

Wir erhalten also den exakten Bruch, so wie gewünscht. Leider hat auch dieses Verfahren einen Nachteil. Betrachten wir die Ausgabe des Programms für den Wert $0,1$:

```
                    0.100000001490116119384765625
                    0.200000002980232238769531250
                    0.400000005960464477539062500
                    0.800000011920928955078125000
                    1.600000023841857910156250000
                    3.200000047683715820312500000
                    6.400000095367431640625000000
                   12.800001907348632812500000000
                   25.600003814697265625000000000
                   51.200007629394531250000000000
                  102.400015258789062500000000000
                  204.800030517578125000000000000
                  409.600061035156250000000000000
                  819.200122070312500000000000000
                 1638.400244140625000000000000000
                 3276.800488281250000000000000000
                 6553.600976562500000000000000000
                13107.201953125000000000000000000
                26214.403906250000000000000000000
                52428.807812500000000000000000000
               104857.615625000000000000000000000
               209715.231250000000000000000000000
               419430.462500000000000000000000000
               838860.925000000000000000000000000
              1677721.850000000000000000000000000
              3355443.700000000000000000000000000
              6710886.400000000000000000000000000
             13421772.800000000000000000000000000
0.1 = 13421773 / 134217728
```

Es ergeben sich außer der initialen Ungenauigkeit bei der Umwandlung ins Binärsystem keine weiteren Ungenauigkeiten. Leider, denn der so ermittelte Bruch ist nur eine Näherung für den Wert $0,1$.

Schauen wir uns noch ein weiteres Beispiel an, bei dem durch Rechenungenauigkeiten bei der Multiplikation mit 10 ein exakter Bruch entsteht. Auch der Wert $2,0815$ kann binär nicht exakt dargestellt werden. Allerdings verschwinden auch hier irgendwann die „überflüssigen“ Nachkommastellen.

```
                    2.081500053405761718750000000
                   20.815000534057617187500000000
                  208.150009155273437500000000000
                 2081.500000000000000000000000000
                20815.000000000000000000000000000
```

Es gilt $2,0815 = (10,000101001101110100101110...)_2$, was bei einer 23 bit großen Mantisse und einem Hidden-Bit zur Zahl $2,08150005340576171875 = (1,00001010011011101001100)_2 \cdot 2^1$

aufgerundet wird.

	1,	0000	1010	0110	1110	1001	100	$[\cdot 2^1]$ $\cdot(1010)_2$
=	1,	0000	1010	0110	1110	1001	100	$[\cdot 2^2]$
+	1,	0000	1010	0110	1110	1001	100	$[\cdot 2^4]$
=	0,	0100	0010	1001	1011	1010	01100	$[\cdot 2^4]$ (Denormalisieren)
+	1,	0000	1010	0110	1110	1001	100	$[\cdot 2^4]$
=	1,	0100	1101	0000	1010	0011	11100	$[\cdot 2^4]$ (Abrunden)
=	1,	0100	1101	0000	1010	0011	111	$[\cdot 2^4]$
=	$20,8150005340576171875$							

Eine weitere Multiplikation mit 10 ergibt:

	1,	0100	1101	0000	1010	0011	111	$[\cdot 2^4]$ $\cdot(1010)_2$
=	1,	0100	1101	0000	1010	0011	111	$[\cdot 2^5]$
+	1,	0100	1101	0000	1010	0011	111	$[\cdot 2^7]$
=	0,	0101	0011	0100	0010	1000	11111	$[\cdot 2^7]$ (Denormalisieren)
+	1,	0100	1101	0000	1010	0011	111	$[\cdot 2^7]$
=	1,	1010	0000	0100	1100	1100	11011	$[\cdot 2^7]$ (Aufrunden)
=	1,	1010	0000	0100	1100	1100	111	$[\cdot 2^7]$
=	$208,1500091552734375$							

Die nächste Multiplikation mit 10 ergibt dann schließlich:

	1,	1010	0000	0100	1100	1100	111	$[\cdot 2^7]$ $\cdot(1010)_2$
=	1,	1010	0000	0100	1100	1100	111	$[\cdot 2^8]$
+	1,	1010	0000	0100	1100	1100	111	$[\cdot 2^{10}]$
=	0,	0110	1000	0001	0011	0011	00111	$[\cdot 2^{10}]$ (Denormalisieren)
+	1,	1010	0000	0100	1100	1100	111	$[\cdot 2^{10}]$
=	10,	0000	1000	0110	0000	0000	00011	$[\cdot 2^{10}]$
=	1,	0000	0100	0011	0000	0000	000011	$[\cdot 2^{11}]$ (Normalisieren)
=	1,	0000	0100	0011	0000	0000	000011	$[\cdot 2^{11}]$ (Abrunden)
=	1,	0000	0100	0011	0000	0000	000	$[\cdot 2^{11}]$
=	$2081,5$							

Die Frage, ob eine Multiplikation mit 2 oder 10 besser ist, ist also gar nicht so einfach zu beantworten. Jedenfalls ist beim Rechnen mit Gleitpunktzahlen Vorsicht geboten. Als letztes soll noch erwähnt werden, dass die Umwandlung der Zahl $3,0078125$ in einen Bruch auch mit dem fortlaufenden Erweitern mit 10 gelingt, wenn man nur den Nachkommateil umwandelt und dann den Vorkommateil addiert. Dazu muss der Vorkommateil auf den gleichen Nenner gebracht werden:

$$3,0078125 = 3 + 0,0078125 = 3 + \frac{1}{128} = \frac{384}{128} + \frac{1}{128} = \frac{385}{128}$$

Hier die Ausgabe des Programms:

```
0.007812500000000000000000000
0.015625000000000000000000000
0.031250000000000000000000000
0.062500000000000000000000000
0.125000000000000000000000000
```

```
                    0.25000000000000000000000000
                    0.50000000000000000000000000
                    1.00000000000000000000000000
0.0078125 = 1 / 128
```

Die obige Abbruchbedingung der While-Schleife in Zeile 5 der Funktion `valueOf` aus Listing 1 kann auch anders formuliert werden, siehe unten, allerdings muss darauf geachtet werden, dass die Datentypen zueinander passen.

```
double d = (1L << 31) + 0.75;
printf("            d = %35.20lf\n", d);

double x = d - (int) d;
printf(" d - (int) d = %35.20lf\n", x);

x = d - (long) d;
printf(" d - (long) d = %35.20lf\n", x);

x = d - trunc(d);
printf("d - trunc(d) = %35.20lf\n", x);
```

Dieses Programm liefert die folgende Ausgabe.

```
            d =     2147483648.75000000000000000000
 d - (int) d =     4294967296.75000000000000000000
d - (long) d =              0.75000000000000000000
d - trunc(d) =              0.75000000000000000000
```

Die zweite Zeile ist so natürlich nicht gewünscht. Eigentlich wollten wir den Nachkommateil von `d` bestimmen, indem wir den Vorkommateil von `d`, also den ganzzahligen Teil subtrahieren. Integer-Werte sind aber auf unserem Rechner nur 32 Bit groß, die Mantisse eines `double`-Wertes hat aber eine Größe von 52 Bit. Große Werte wie $2^{31} + 0,75$ können also mit dem Typ `int` nicht dargestellt werden. Daher ist es hier erforderlich, entweder auf den 64-Bit großen Datentyp `long` zu wechseln, oder mittels der Funktion `trunc` den Nachkommateil abzuschneiden.