B.5 Floating point numbers

Solution to Exercise 56.

	type	value	lavlue/rvalue
a)	`bool`	`true`	rvalue
b)	`int`	5	rvalue
c)	`int`	0	lvalue
d)	`double`	0.5	rvalue
e)	`int`	2	rvalue
f)	`int`	2	lvalue

Solution to Exercise 57.

	type	value
(a)	`bool`	`false`
(b)	`double`	0.5
(c)	`float`	0.5
(d)	`bool`	`true`
(e)	`bool`	`false`
(f)	`int`	11
(g)	`int`	1
(h)	`bool`	`false`
(i)	`bool`	`true`
(j)	`double`	5.5

Solution to Exercise 58.

a) `6 / 4 * 2.0f - 3` $\longrightarrow$
`1 * 2.0f - 3` $\longrightarrow$
`2.0f - 3` $\longrightarrow$
`-1.0f`

b) `2 + 15.0e7f - 3 / 2.0 * 1.0e8` $\overset{1.5\cdot 10^8 > 2^{27}}{\longrightarrow}$
`15.0e7f - 3 / 2.0 * 1.0e8` $\longrightarrow$
`15.0e7f - 1.5 * 1.0e8` $\longrightarrow$
`15.0e7f - 1.5e8` $\longrightarrow$
`0.0`

c) `392593 * 2735.0f - 8192 * 131072 + 1.0` $\longrightarrow$
binary: `1`$\underbrace{\texttt{0...0}}_{\text{23 times}}\underbrace{\texttt{0011111}}_{\text{get lost}}$`f - 8192 * 131072 + 1.0` $\longrightarrow$
`1073741824.0f - 8192 * 131072 + 1.0` $\longrightarrow$
`1073741824.0f - 1073741824 + 1.0` $\longrightarrow$

```
0.0f + 1.0 ⟶
1.0
```

```
d) 16 * (0.2f + 262144 - 262144.0) ⟶
16 * (binary:1 0...0 .00110 0110f - 262144.0) ⟶
          18 times   get lost
16 * binary:0.0011 ⟶
3.0
```

Solution to Exercise 59.

a) This is easy and doesn't take any calculations: $0.25 = 1/4 = 1 \cdot 2^{-2}$. As a binary number, this is 0.01.

b) We employ the rules from Section 2.5.5.

$$
\begin{array}{rcrcrcl}
 & & & & 1.52 & \to & b_0 = 1 \\
2(1.52-1) & = & 2 \cdot 0.52 & = & 1.04 & \to & b_{-1} = 1 \\
2(1.04-1) & = & 2 \cdot 0.04 & = & 0.08 & \to & b_{-2} = 0 \\
2(0.08-0) & = & 2 \cdot 0.08 & = & 0.16 & \to & b_{-3} = 0 \\
2(0.16-0) & = & 2 \cdot 0.16 & = & 0.32 & \to & b_{-4} = 0 \\
2(0.32-0) & = & 2 \cdot 0.32 & = & 0.64 & \to & b_{-5} = 0 \\
2(0.64-0) & = & 2 \cdot 0.64 & = & 1.28 & \to & b_{-6} = 1 \\
2(1.28-1) & = & 2 \cdot 0.28 & = & 0.56 & \to & b_{-7} = 0 \\
2(0.56-0) & = & 2 \cdot 0.56 & = & 1.12 & \to & b_{-8} = 1 \\
2(1.12-1) & = & 2 \cdot 0.12 & = & 0.24 & \to & b_{-9} = 0 \\
2(0.24-0) & = & 2 \cdot 0.24 & = & 0.48 & \to & b_{-10} = 0 \\
2(0.48-0) & = & 2 \cdot 0.48 & = & 0.96 & \to & b_{-11} = 0 \\
2(0.96-0) & = & 2 \cdot 0.96 & = & 1.92 & \to & b_{-12} = 1 \\
2(1.92-1) & = & 2 \cdot 0.92 & = & 1.84 & \to & b_{-13} = 1 \\
2(1.84-1) & = & 2 \cdot 0.84 & = & 1.68 & \to & b_{-14} = 1 \\
2(1.68-1) & = & 2 \cdot 0.68 & = & 1.36 & \to & b_{-15} = 1 \\
2(1.36-1) & = & 2 \cdot 0.36 & = & 0.72 & \to & b_{-16} = 0 \\
2(0.72-0) & = & 2 \cdot 0.72 & = & 1.44 & \to & b_{-17} = 1 \\
2(1.44-1) & = & 2 \cdot 0.44 & = & 0.88 & \to & b_{-18} = 0 \\
2(0.88-0) & = & 2 \cdot 0.88 & = & 1.76 & \to & b_{-19} = 1 \\
2(1.76-1) & = & 2 \cdot 0.76 & = & 1.52 & \to & b_{-20} = 1 \\
 & & & & \vdots & &
\end{array}
$$

Phew, finally the sequence becomes periodic, and we get the binary expansion $1.\overline{10000101000111101011}$.

c) We employ the rules from Section 2.5.5.

$$\begin{array}{rcl} & & 1.3 \rightarrow b_0 = 1 \\ 2(1.3-1) = 2\cdot 0.3 & = & 0.6 \rightarrow b_{-1} = 0 \\ 2(0.6-0) = 2\cdot 0.6 & = & 1.2 \rightarrow b_{-2} = 1 \\ 2(1.2-1) = 2\cdot 0.2 & = & 0.4 \rightarrow b_{-3} = 0 \\ 2(0.4-0) = 2\cdot 0.4 & = & 0.8 \rightarrow b_{-4} = 0 \\ 2(0.8-0) = 2\cdot 0.8 & = & 1.6 \rightarrow b_{-5} = 1 \\ 2(1.6-1) = 2\cdot 0.6 & = & 1.2 \rightarrow b_{-6} = 1 \\ & & \vdots \end{array}$$

We see that the expansion is periodic and yields the binary number $1.0\overline{1001}$.

d) We write $11.1 = 10 + 1.1$ and add the binary expansion 1010.0 of 10 to the binary expansion $1.0\overline{0011}$ of 1.1 derived in Section 2.5.5. The resulting expansion is $1011.0\overline{0011}$.

Solution to Exercise 60.

a) 0.25 has normalized binary floating point representation $1.0 \cdot 2^{-2}$ and is therefore smaller than any number in $\mathcal{F}^*(2,5,-1,2)$. The nearest number is therefore the smallest number in this system, namely 0.5 with normalized binary representation $1.0 \cdot 2^{-1}$. In $\mathcal{F}(2,5,-1,2)$, we can represent 0.25 exactly as $0.1 \cdot 2^{-1}$.

b) 1.52 has normalized binary floating point representation $1.\overline{10000101000111101011} \cdot 2^0$. To get the nearest number in $\mathcal{F}^*(2,5,-1,2)$, we have to round to 5 significant digits. The result is $1.1000 \cdot 2^0 = 1.5$, obtained by rounding down, since $1.1001 \cdot 2^0 = 1.5625$, obtained by rounding up, is farther away. The nearest number in $\mathcal{F}(2,5,-1,2)$ is the same, since this system has only extra numbers *smaller* than any normalized number. Such numbers cannot be nearest to numbers *larger* than some normalized number.

c) 1.3 has normalized binary floating point representation $1.0\overline{1001}$. To get the nearest number in $\mathcal{F}^*(2,5,-1,2)$, we have to round to 5 significant digits. The result is $1.0101 \cdot 2^0 = 1.3125$, obtained from rounding up, since $1.0100 \cdot 2^0 = 1.25$, obtained from rounding down, is farther away. The nearest number in $\mathcal{F}(2,5,-1,2)$ is the same.

d) 11.1 is larger than any number in the system $\mathcal{F}^*(2,5,-1,2)$. Recall that the largest number is $1.1111 \cdot 2^2 = 4+2+1+1/2+1/4 = 7.75$, and this is the nearest number to 11.1, also in $\mathcal{F}(2,5,-1,2)$.

Solution to Exercise 61. The smallest normalized number is always $2^{e_{min}}$. In case of single precision, this is 2^{-126}, for double precision, it is 2^{-1022}. Recall that the largest normalized number is

$$\left(1 - \left(\frac{1}{\beta}\right)^{p}\right) \beta^{e_{max}+1}.$$

For single precision, this yields

$$\left(1 - \left(\frac{1}{2}\right)^{24}\right) 2^{128} = 2^{128} - 2^{104}.$$

For double precision, we get

$$\left(1 - \left(\frac{1}{2}\right)^{53}\right) 2^{1024} = 2^{1024} - 2^{971}.$$

Solution to Exercise 62. For each exponent, $\mathcal{F}^*(\beta, p, e_{min}, e_{max})$ has $\beta - 1$ possibilities for the first digit, and β possibilities for the remaining $p - 1$ digits. The size of $\mathcal{F}^*(\beta, p, e_{min}, e_{max})$ is therefore

$$2(e_{max} - e_{min} + 1)(\beta - 1)\beta^{p-1},$$

if we take the two possible signs into account.

$\mathcal{F}(\beta, p, e_{min}, e_{max})$ has extra nonnegative numbers of the form

$$0.d_1 \dots d_{p-1} 2^{e_{min}},$$

and there are β^{p-1} of them. Adding the non-positive ones and subtracting 1 for counting 0 twice, we get

$$2\beta^{p-1} - 1$$

extra numbers.

Solution to Exercise 63. The binary expansion of 0.1 is $0.0\overline{0011}$, obtained from the representation of 1.1 by subtracting 1. This value has to be rounded to the nearest value with 24 significant digits. Let us write out the expansion so that we get the first 26 significant digits of $0.0\overline{0011}$:

$$0.00011001100110011001100110011.$$

It follows that we have to round up to 1 at digit 24 to get the nearest `float` value

$$1.10011001100110011001101 \cdot 2^{-4}.$$

To see how this value differs from 0.1, let's convert it back into decimal representation. Interestingly, this is always possible without any error, since 0.1 (binary) is 0.5 (decimal), 0.01 (binary) is 0.25 (decimal), and so on. The decimal value that we obtain is

0.100000001490116119384765625.

Solution to Exercise 64. We compare floating point numbers for equality in `i != 1.0`, although one of them (namely the value of `i`) is the result of inexact computations, assuming a base-2 floating point number system. The inexactness comes from the rounding of 0.1 to a floating point number, and from the subsequent addition of numbers. In practice, this leads to an infinite loop, since `i != 1.0` will always be satisfied.

Solution to Exercise 65. The problem is that this is an infinite loop. Here is why: The literal `100000000.0f` has as its value the nearest `float` value to 10^8. What is this value? It *is* 10^8: the binary representation of 10^8 is 101111101011110000100000000 (as you can for example see by running program `dec2bin2.cpp`, see Exercise 50). This has 19 significant digits and therefore easily fits into a `float` value. But the total number of binary digits of 10^8 is 27. Now, the loop tries to run through all numbers up to 10^8. This works as long as the numbers have 24 or less binary digits in total, since they are exactly representable as `float` values in this case.

But at some point, the loop reaches the number 2^{24} which is the first one with a 25-digit binary representation, namely 1000000000000000000000000. This value is still representable over the type `float`, since the number of significant digits is 1. The next number is $2^{24}+1$ with binary representation 1000000000000000000000001. This is still a number with 25 binary digits, but now all of them are significant. A `float` value can accommodate only 24 significant digits, and the value is therefore being rounded to the next representable number. In this case, there are two such numbers, 2^{24} and $2^{24}+2$, and by the round-to-even rule, 2^{24} is chosen.

Thus, the loop does not make progress beyond 2^{24} which means that it never terminates. The following program checks for the first repetition in the value of `i`.

```
1  // demonstrates the effect of floating-point arithmetic on
2  // a simple loop
3
4  #include<iostream>
5
6  int main()
7  {
8    // we suspect that i does not grow beyond 2^24,
9    // so we check whether this is really the case
10   float old_i = -1.0f;
11   for (float i = 0.0f; i < 100000000.0f; ++i)
12     if (i == old_i) {
13       // infinite loop!
14       std::cout << "loop gets stuck with i = " << int(i) << ".\n";
15       break;
16     } else
17       old_i = i;
18   return 0;
19 }
```

On the authors' platform, the first repeating value is indeed 2^{24}:

```
loop gets stuck with i = 16777216.
```

Solution to Exercise 66.

```
// Prog: dec2float.cpp
// compute the float representation of a number
// in the open interval (0,2)

#include <iostream>

int main ()
{
  // input
  std::cout << "Decimal number x (0 < x < 2) =? ";
  float x;
  std::cin >> x;

  // x = w * 2^e
  float w = x;
  int e = 0;

  // as long as w < 1, decrement e and double w
  for ( ; w < 1.0f; w *= 2.0f) --e;

  // Now we have 1 <= w < 2, apply rule from lecture
  std::cout << "Significand: ";
  for ( ; w != 0.0; w = 2.0f * (w - int(w)))
    std::cout << int(w);

  std::cout << "\nExponent: " << e << "\n";

  return 0;
}
```

Solution to Exercise 67.

```
// Prog: double_integer.cpp
// tests whether a given double value is integer
//
#include<iostream>

int main()
{
  // input
  std::cout << "Decimal number =? ";
  double d;
  std::cin >> d;
  const double abs_d = d > 0 ? d: -d; // |d|

  // |d| can be written in the form m * 2^e, where m is a
  // natural number whose last binary digit is 1. Then |d| is
  // integer if and only if e>=0. Having m, we can therefore
  // conclude that d is integer if and only if |d| >= m

  // step 1: normalize such that number is in [1,2)
  double dd = abs_d;
  while (dd >=2) dd/=2;  // ensure dd < 2;
  while (dd < 1) dd*=2;  // ensure dd >=1;
```

```
23
24    // step 2: compute binary expansion m like in the lecture notes
25    double m = 0;
26    while (dd > 0) {
27      // move last binary digit of dd into m
28      m *= 2;
29      if (dd >= 1) {
30        m += 1;
31        dd = 2*(dd-1);
32      } else
33        dd = 2*dd;
34    }
35
36    // step 3: compare with abs_d
37    std::cout << d;
38    if (abs_d >= m)
39      std::cout << " is integer.\n";
40    else
41      std::cout << " is not integer.\n";
42
43    return 0;
44  }
```

Solution to Exercise 68. Here is the program based on the first formula.

```
1   // Prog: pi1.cpp
2   // approximate pi according to first n terms of the formula
3   // pi = 4 - 4/3 + 4/5 - 4/7 ...
4
5   #include <iostream>
6
7   int main ()
8   {
9     // input
10    std::cout << "Number of iterations =? ";
11    unsigned int n;
12    std::cin >> n;
13
14    // computation (forward sum)
15    double pif = 0.0;
16    for (int i = 1; i < 2*n; i += 2)
17      if (i % 4 == 1)
18        pif += 4.0 / i;
19      else
20        pif -= 4.0 / i;
21
22    // computation (backward sum)
23    double pib = 0.0;
24    for (int i = 2*n-1; i > 0; i -= 2)
25      if (i % 4 == 1)
26        pib += 4.0 / i;
27      else
28        pib -= 4.0 / i;
29
30    // output
31     std::cout << "Pi is approximately "
32               << pif << " (forward sum), or "
33               << pib << " (backward sum); the difference is "
34               << pif - pib << "\n";
35
36    return 0;
37  }
```

When you run it for $n = 10,000$, for example, it gives on our platform the approximation 3.14139 (still off in the fourth digit after the decimal point). For $n = 100,000$, we get 3.14157 (still off in the fifth digit after the decimal point). For $n = 1,000,000$, finally, the result is correct to five digits after the decimal point: 3.14159.

Here is the approximation based on the second formula.

```
1  // Prog: pi2.cpp
2  // approximate pi according to the first n terms of the formula
3  // pi = 2 + 2*1 / 3 + 2*1*2 / 3*5 + 2*1*2*3 / 3*5*7
4
5  #include <iostream>
6
7  int main ()
8  {
9    // input
10   std::cout << "Number of iterations =? ";
11   unsigned int n;
12   std::cin >> n;
13
14   // auxiliary variables
15   // initialized for first term of forward sum (i=0)
16   double numer = 2.0; // numerator i-th term
17   double denom = 1.0; // denominator i-th term
18
19   // forward sum
20   // pif: value after term i (i=0 initially, then i=1,2,...,n-1)
21   double pif = 2.0;
22   for (int i = 1; i < n; ++i)
23     pif += (numer *= i) / (denom *= (2*i + 1)); // update to term i
24   // now numer and denom are the ones for i=n-1
25
26   // backward sum
27   // pib: value after term i (i=n-1 initially, then i=n-2,...,1,0)
28   double pib = numer / denom;
29   for (int i = n-1; i >= 1; --i) {
30     pib += (numer /= i) / (denom /= (2*i + 1)); // update to term i-1
31   }
32
33   // output
34   std::cout << "Pi is approximately "
35             << pif << " (forward sum), or "
36             << pib << " (backward sum); the difference is "
37             << pif - pib << "\n";
38
39   return 0;
40 }
```

This already gives the result 3.14159 for $n = 17$ on our platform, so this version is obviously preferable.

Solution to Exercise 69. Here is the program, including the function(s) and a test; for large x, the idea is to use periodicity of $\sin(x)$, meaning that $\sin(x) = \sin(x - 2\pi)$ for all x. This means, we can reduce the argument modulo 2π to make it small, and only then call the function `sinus`.

```
1  // Prog: sinus.cpp
2  // implements and tests functions for computing sin(x)
3  #include<iostream>
```

```
 4
 5  // POST: returns an approximation of sin(x)
 6  double sinus (double x) {
 7     double result = x;
 8     double temp = x;
 9     for (int i = 1; i < 20; ++i) {
10        temp *= x / (2*i+1) * x / (2*i);
11        if (i % 2 == 0) {
12          result += temp;
13        } else {
14          result -= temp;
15        }
16     }
17
18     return result;
19  }
20
21  // POST: returns an approximation of sin(x)
22  double sinus2  (double x) {
23    // before we start, reduce x modulo 2pi. The following is not a clean
24    // solution since it overflows for large x, but it shows the idea.
25    double pi = 3.1415926535;
26    int y = int(x / (2*pi));
27    x -= y*2*pi;
28    return sinus(x);
29  }
30
31  // test function
32  void test_sinus (double x) {
33    std::cout << "sin (" << x << ") = "
34              << sinus(x) << " ; " << sinus2(x) << "\n";
35  }
36
37  // now test it
38  int main() {
39    double pi = 3.1415926535;
40    test_sinus (0);                    // should be 0
41    test_sinus (pi/6);                 // 30 degrees; should be 1/2
42    test_sinus (pi/4);                 // 45 degrees; should be 1/2 * sqrt(2)
43    test_sinus (pi/3);                 // 60 degrees; should be 1/2 * sqrt(3)
44    test_sinus (pi/2);                 // should be 1
45    test_sinus (pi/2 + 10000*2*pi);    // should be 1, due to periodicity
46
47  }
```

Solution to Exercise 70.

```
 1  // Program: babylonian.cpp
 2  // Approximation of the square root of a positive real number
 3
 4  #include <iostream>
 5
 6  int main(){
 7
 8      // Read input
 9      double s; // input number
10      const double eps = 0.001; // the epsilon, i.e. max square error
11
12      std::cout << "Which number do you want to take the square root of?";
13      std::cout << "\n";
14      std::cin >> s;
15
16      // Compute square root
```

```
17      double x = s / 2.0; // initialize solution
18      unsigned int n = 0; // counter for number of iterations
19
20      while (x * x - s >= eps || s - x * x >= eps) {
21          ++n;
22          x = (x + s / x) / 2.0;
23      }
24
25      std::cout << "The square root of " << s << " is: " << x << std::endl;
26      std::cout << "The number of iterations done: " << n << std::endl;
27
28      return 0;
29  }
```

Solution to Exercise 71.

```
1   // Program: fpsys.cpp
2   // Provide a graphical representation of floating point numbers
3
4   #include <iostream>
5   #include <IFM/window>
6
7   int main()
8   {
9     // Input parameters of floating point system
10    std::cout << "Draw F*(2,p,e_min,e_max).\np =? ";
11    unsigned int p;
12    std::cin >> p;
13    std::cout << "e_min =? ";
14    int emin;
15    std::cin >> emin;
16    std::cout << "e_max =? ";
17    int emax;
18    std::cin >> emax;
19
20    // We compute significands using integral arithmetic, that is,
21    // scaled by 2^(p-1).
22
23    // compute the smallest normalized significand 2^(p-1)
24    unsigned int smin = 1;
25    for (unsigned int i = 1; i < p; ++i) smin *= 2;
26    // compute the largest normalized significand (2^p)-1
27    const unsigned int smax = 2 * smin - 1;
28    // compute 2^emin
29    double pemin = 1;
30    for (int i = 0; i < emin; ++i) pemin *= 2;
31    for (int i = 0; i > emin; --i) pemin /= 2;
32    // compute 2^emax
33    double pemax = 1;
34    for (int i = 0; i < emax; ++i) pemax *= 2;
35    for (int i = 0; i > emax; --i) pemax /= 2;
36
37    // For each positive number x of the system draw a circle
38    // with radius x around the window center
39
40    // parameters to scale output
41    const int cx = (ifm::wio.xmax() - ifm::wio.xmin()) / 2;
42    const int cy = (ifm::wio.ymax() - ifm::wio.ymin()) / 2;
43    const double scale = cx / (pemax * smax);
44
45    // zero
46    ifm::wio << ifm::Point(cx, cy);
47    // loop over all normalized significands
```

```
48    for (unsigned int i = smin; i <= smax; ++i)
49      // loop over all exponents
50      for (double m = pemin; m <= pemax; m *= 2)
51        ifm::wio << ifm::Circle(cx, cy, int(m * i * scale));
52
53    ifm::wio.wait_for_mouse_click();
54    return 0;
55  }
```

Solution to Exercise 72. Mr. Plestudent is wrong. When we use the algorithm for constructing the hexadecimal representation, we see a periodic pattern, just as for $\beta = 2$.

$$\begin{array}{rcrcrcl} & & & & 0.1 & \to & b_0 = 0 \\ 16(0.1-0) & = & 16\cdot 0.1 & = & 1.6 & \to & b_{-1} = 1 \\ 16(1.6-1) & = & 16\cdot 0.6 & = & 9.6 & \to & b_{-2} = 9 \\ 16(9.6-9) & = & 16\cdot 0.6 & = & 9.6 & \to & b_{-3} = 9 \\ & & & & \vdots & & \end{array}$$

Somewhat less formally, we can also argue as follows: the hexadecimal representation is obtained from the binary one by repeatedly grouping 4 consecutive binary digits in $\{0,1\}$ into one hexadecimal digit in $\{0,1,\ldots,9,A,B,\ldots,F\}$. Then, if the binary expansion is infinite, so will be the hexadecimal one.

Again a bit more formal for the careful reader: Let's assume we have a representation

$$\frac{1}{10} = \sum_{i=1}^{p} d_i 16^{-i}.$$

We can write each d_i as a four-digit binary number:

$$d_i = d_{i_3}2^3 + d_{i_2}2^2 + d_{i_1}2^1 + d_{i_0}2^0, \quad d_{i_3}, d_{i_2}, d_{i_1}, d_{i_0} \in \{0,1\}.$$

We then get

$$\frac{1}{10} = \sum_{i=1}^{p} d_i 16^{-i} = \sum_{i=1}^{p} d_i 2^{-4i} = \sum_{i=1}^{p} (\sum_{j=0}^{3} d_{i_j} 2^j) 2^{-4i} = \sum_{i=1}^{p} \sum_{j=0}^{3} d_{i_j} 2^{j-4i},$$

and this is a finite binary floating-point number, in contradiction to the known fact there is no such number.

Solution to Exercise 72. Because of

$$2^{-i} = 5^i \cdot 10^{-i},$$

we have

$$b = \sum_{i=1}^{k} b_i 2^{-i} = \sum_{i=1}^{k} (b_i 5^i) 10^{-i},$$

where all values $b_i 5^i$ are natural numbers. Thus, b is a finite sum of numbers of the form 10^{-i} that obviously have (very simple) finite decimal representations. But then the sum also has finite representation (just add the numbers up, using elementary school arithmetic).

Solution to Exercise 74. Here is the characterization. γ refines β if and only if all prime factors of β are also prime factors of γ. For example, 10 refines 2, since 2 is also a prime factor of 10. But 2 does not refine 10, since 5 is a prime factor of 10 but not of 2. Maybe surprisingly, a smaller number may refine a larger number, for example, 2 refines 16.

To prove the characterization, we need to show two directions. Let us first assume that all prime factors of β are also prime factors of γ. Now consider any number

$$r = s \cdot \sum_{i=0}^{p-1} d_i \beta^{-i} \cdot \beta^e \in \mathcal{F}(\beta, p, e_{\min}, e_{\max}),$$

i.e. $s \in \{-1, 1\}$, $d_i \in \{0, \ldots, \beta - 1\}$ for all i, and $e \in \{e_{\min}, \ldots, e_{\max}\}$. Since all prime factors of β are also prime factors of γ, there is some integer power γ^M of γ that is a multiple of β (choose M such that every prime factor of γ^M has at least the multiplicity that it has in β). So we have an integer m with

$$\beta \cdot m = \gamma^M,$$

which implies that

$$\beta^{-i} \cdot \beta^e = \beta^{e-i} = \begin{cases} \beta^{e-i}\gamma^0, & \text{if } e - i \geq 0, \\ m^{i-e}\gamma^{M(e-i)}, & \text{if } e - i < 0. \end{cases}$$

This means that r is a finite sum of integer powers of γ. Adding up all these powers as we do in elementary school shows that $r \in \mathcal{F}(\gamma, q, f_{\min}, f_{\max})$ for suitable $q, f_{\min}, f_{\max}$. Since $\mathcal{F}(\beta, p, e_{\min}, e_{\max})$ contains only finitely many numbers, we can choose $q, f_{\min}, f_{\max}$ that work for all of them, and we are done.

For the other direction, let us now assume that there is a prime factor t of β that is not a prime factor of γ. We will show that there is a number in $\mathcal{F}(\beta, 2, 0, 0)$ that is not contained in any system of the form $\mathcal{F}(\gamma, q, f_{\min}, f_{\max})$. This number is simply β^{-1}. Suppose for a contradiction that we could write β^{-1} in the form

$$\beta^{-1} = \sum_{i=0}^{q-1} d_i \gamma^{-i} \cdot \gamma^f.$$

Multiplying the equation with a sufficiently large power γ^M of γ makes the right-hand side (and thus also the left-hand side) integer. Hence,

$$\frac{\gamma^M}{\beta}$$

is integer, so γ^M is a multiple of β. This, however, is a contradiction to γ not containing prime factor t.

Solution to Exercise 75.

```
// Prog: mandelbrot.cpp
// draws (a part of) the Mandelbrot set and allows the user to
// zoom in by clicking with the mouse on the region to be enlarged
//
// The Mandelbrot set is defined as the set of all complex numbers
// c such that the complex iteration formula z := z^2 + c (starting
// with z=0) always yields values z of absolute value at most two.
// In the computations below, we perform a large but fixed number
// of steps of this iteration for a given c; if all computed values
// are at most two in absolute value, we consider c as part of the
// Mandelbrot set (and depict its corresponding pixel in black),
// otherwise we draw a white pixel.

#include<IFM/window>

int main()
{
  // the currently considered subset of the complex plane, initially
  // [-2, 1] x [-1, 1] (covers the so-called main cardioid of the
  // Mandelbrot set)
  double r_min = -2; double r_max = 1;
  double i_min = -1; double i_max= 1;

  // window scaling factor; change this for larger/smaller display
  // window
  double window_scale = 500;

  // zoom factor from one iteration to the next
  double zoom_factor = 10;

  // the display window dimensions in pixels (window should be
  // congruent to the current complex plane subset)
  int x_size = int (window_scale * (r_max - r_min));
  int y_size = int (window_scale * (i_max - i_min));

  // open the display window
  ifm::Wstream w (x_size, y_size,
                  "The Mandelbrot set (click to zoom in)");

  // maximum number of iterations (the higher, the more accurate;
  // the lower, the faster)
  unsigned int max_iter = 500;

  // main drawing loop; one iteration for every zoom scale
  for(;;) {
    // go through all pixels
    for (int x=0; x<x_size; ++x)
      for (int y=0; y<y_size; ++y) {

        // compute corresponding point in complex plane
        const double r = r_min + x / window_scale;
        const double i = i_min + y / window_scale;

        // do the Mandelbrot iteration for that point
        // interpreted as complex number c = (r,i)
        double r_z = 0; // z (real part)
        double i_z = 0; // z (imaginary part)
        unsigned int iter = 0;
```

```
59          while (iter < max_iter && r_z * r_z + i_z * i_z <= 4) {
60            // |z| <= 2; replace z by z^2 + c
61            double h = r_z * r_z - i_z * i_z + r; // new z_r
62            i_z = 2 * r_z * i_z + i;              // new z_i
63            r_z = h;
64            ++iter;
65          }
66          // coloring: max_iter -> black, other -> white
67          if (iter == max_iter)
68            w.set_color (w.number_of_colors()-2); // black
69          else
70            w.set_color (w.number_of_colors()-1); // white
71          w << ifm::Point (x, y);
72        }
73
74      // zoom in; new center is mouse click position
75      int x_c; int y_c;
76      w.get_mouse_click (x_c, y_c);
77      const double r_c = r_min + x_c / window_scale;
78      const double i_c = i_min + y_c / window_scale;
79      const double r_span = r_max - r_min;
80      const double i_span = i_max - i_min;
81      r_min = r_c - 0.5 * r_span / zoom_factor;
82      r_max = r_c + 0.5 * r_span / zoom_factor;
83      i_min = i_c - 0.5 * i_span / zoom_factor;
84      i_max = i_c + 0.5 * i_span / zoom_factor;
85      window_scale *= zoom_factor;
86      w.clear();
87    }
88
89    return 0;
90  }
```

Solution to Exercise 76. CGAL is the *Computational Geometry Algorithms Library*, an open source C++ library of data structures and algorithms for solving geometric problems. The CGAL homepage is `www.cgal.org`.

`CGAL::orientation` is a function that determines for three given points $p, q, r \in \mathbb{R}^2$ whether r lies to the left, on, or to the right of the oriented line through p and q. The resulting values (`CGAL::LEFTTURN`, `CGAL::COLLINEAR`, or `CGAL::RIGHTTURN`) define the orientation of the point triple $\{p, q, r\}$. `CGAL::LEFTTURN` means that p, q, r appear in counterclockwise order around the triangle spanned by p, q, r, while `CGAL::RIGHTTURN` signals clockwise order. `CGAL::COLLINEAR` means that all three points are on a common line, so the triangle is "flat".

The writer of the email is surprised since the orientation of a point triple should not change when all point coordinates are multiplied with a fixed scalar (in this case 100). But in reality, it does change, at least according to the function `CGAL::orientation`.

The reason is that the integer coordinates of the points $(14, 22), (15, 21), (19, 17)$ can be converted to `float` or `double` (we don't exactly know which of the two the writer of the email is using) without any error. In contrast, some of the coordinates of $(0.14, 0.22), (0.15, 0.21), (0.19, 0.17)$ don't have finite binary representations, so in converting them to `float` or `double`, errors are inevitable. Since the points are mathematically collinear (on the same line), it is clear that the tiniest error is enough to destroy this property. That's why `CGAL::orientation` delivers a result different from

CGAL::COLLINEAR.

Here is what you could answer the writer of the email.

```
Hi,

assuming that you use type float or double to represent the point
coordinates, the inconsistency that you reported is due to the
conversion of point coordinates from the decimal input format to the
internally used binary format. Decimal integers like 14, 22 etc. can be
represented exactly in binary format, and CGAL::orientation returns
the correct answer CGAL::COLLINEAR for the three points with integer
coordinates. But decimal fractions like 0.14, 0.22 etc. do not
necessarily have finite representations in binary format. This is like
trying to write the number 1/3 as a decimal fraction. The best you can
do is 0.33333... but wherever you stop, you make a small error.

Now, CGAL::orientation sees the points (0.14, 0.22), (0.15, 0.21) and
(0.19,0.17) only after the conversion to binary format, and this
conversion introduces some (tiny) errors. But since the points are
mathematically collinear, even the tiniest errors may have the effect
of destroying collinearity. This is exactly what you observed.

The problem is inevitable in working with floating-point numbers, since
you cannot circumvent the decimal-to-binary conversion. All you can do
is to only use point coordinates (integers, for example) for which the
conversion is exact.
```