Solution 1

a) Yes it is an element, since its (unique) normalized binary floating point representation is $1.10 * 2^0$ (as can be seen for example by applying the algorithm from the lecture) whose significand only needs 2 binary digits while $F^*(2, 3, -1, 2)$ even allows for three. And furthermore, the exponent is in the allowed range: $0 \in \{-1, 0, 1, 2\}$.

b) The largest number in binary is $1.11 * 2^2$ which is $1.75 * 4 = 7$ in decimal.

c) We first notice two things from the lecture that make our lives easier here: (i) in normalized floating point systems all numbers are uniquely representable, and (ii) for every number in the system also its negative counterpart is in the system. Thus by (ii) the total number is twice the number of positive numbers, thus we can just count these:

A positive number in the system is of the form $1.xy * 2^e$, which implies that for a fixed exponent there are four possible significands ($1.00, 1.01, 1.10, 1.11$) since we can vary $x$ and $y$ independently but cannot vary the first 1. On the other hand, the system allows for four possible exponents $e$ ($-1, 0, 1, 2$) and because of (i) we don’t count any number twice.

Combining these observations we obtain a total of $2 * 4 * 4 = 32$ numbers.

d) $a==1$. The conversion of a non-negative float value $a$ to the type unsigned int corresponds to the $\lfloor x \rfloor$ operator.

e) First of all, notice that * and / have the same precedence and are left-associative. Thus the expression is parenthesized as $(4.0 * 5) / 8)$. Therefore, $4.0 * 5$ has to be computed first, which is $20.0$ and the $20.0 / 8$ is computed which is $2.5$. And the type of the expression is double.

Solution 2

a) We use the algorithm from the lecture.
\[
\begin{array}{cccc}
x & b_i & x - b_i & 2(x - b_i) \\
0.3 & 0 & 0.3 & 0.6 \\
0.6 & 0 & 0.6 & 1.2 \\
1.2 & 1 & 0.2 & 0.4 \\
0.4 & 0 & 0.4 & 0.8 \\
0.8 & 0 & 0.8 & 1.6 \\
1.6 & 1 & 0.6 & 1.2 \\
\end{array}
\]

Thus the resulting binary representation is: 0.01001

b) We can translate 11 and 0.7 separately. We can do this since if \(a_{dec} = b_{dec} + c_{dec}\) holds for decimal numbers \(a_{dec}\), \(b_{dec}\), \(c_{dec}\), then there obviously must hold \(a_{bin} = b_{bin} + c_{bin}\) for their binary representations \(a_{bin}\), \(b_{bin}\), \(c_{bin}\).

Case 11: This is 1011. (The algorithm for this was for example applied in sheet 3, exercise 3.)

Case 0.7: We use the algorithm from the lecture.

\[
\begin{array}{cccc}
x & b_i & x - b_i & 2(x - b_i) \\
0.7 & 0 & 0.7 & 1.4 \\
1.4 & 1 & 0.4 & 0.8 \\
0.8 & 0 & 0.8 & 1.6 \\
1.6 & 1 & 0.6 & 1.2 \\
1.2 & 1 & 0.2 & 0.4 \\
0.4 & 0 & 0.4 & 0.8 \\
\end{array}
\]

Thus the resulting binary representation is: 0.10110.

Case 11.7: Since 11.7 = 11 + 0.7 we get 1011.10110 as the binary representation of 11.7.

c) The table can be obtained by proceeding as described on the lecture handout 5, PDF-page 14:

\[
\begin{array}{cc}
(0.25 + 0.25) + 4 & (4 + 0.25) + 0.25 \\
\hline
\text{decimal} & \text{binary} & \text{decimal} & \text{binary} \\
0.25 & 1.00 \times 2^{-2} & 4 & 1.00 \times 2^2 \\
+ 0.25 & 1.00 \times 2^{-2} & + 0.25 & 1.00 \times 2^{-2} \\
= & 1.00 \times 2^{-1} & = & 1.00 \times 2^2 (\text{(*)}) \\
+ 4 & 1.00 \times 2^2 & + 0.25 & 1.00 \times 2^{-2} \\
= 5 & 1.01 \times 2^2 & = 4 & 1.00 \times 2^2 \\
\end{array}
\]

As an example we compute (\text{(*)}) in more detail. To this end, think back of the four steps seen in the lecture:

a) Bring both numbers to the same exponent. We can pick for example the exponent \(-2\). Thus \(10000 \times 2^{-2}\) and \(1 \times 2^{-2}\).
b) Add the significands as binary numbers:

\[
\begin{align*}
10000 \times 2^{-2} \\
+ 1 \times 2^{-2} \\
\hline
10001 \times 2^{-2}
\end{align*}
\]

c) Re-normalize the sum: \( 1.0001 \times 2^2 \)

d) Round if necessary: \( 1.00 \times 2^2 \)

Solution 3

```cpp
// Informatik - Serie 5 - Aufgabe 3
// Programm: point_on_line.cpp
// Autor: ... (Gruppe ...)
// Determines for input coordinates whether the corresponding point
// lies on the line \( g(x) = 2.1x + 0.5 \)

#include "tests.h"
#include <iostream>

// POST: returns |x|
double abs (double x) {
    // Note: As soon as you've seen std::abs you might want to
    // consider using it instead of this function.
    if (x < 0) x *= -1;
    return x;
}

// POST: returns true iff |x-y| < 0.000001
bool are_equal (double x, double y) {
    return abs(x-y) < 0.000001;
}

int main()
{
    // Input coordinates
    double x;
    std::cin >> x;
    double y;
    std::cin >> y;

    // Determine whether (x,y) is on the line
    if (are_equal(2.1*x + 0.5, y))
        std::cout << "Point is on line\n";
}```
Solution 4

One can show that for the given PRE-condition the difference between the truncated number and the original number is always exactly representable as a double. Now we can exactly test whether the error is less than 0.5 (in which case we return the truncation), or at least 0.5 (in which case we return the next integer, going away from zero).

```cpp
#include <iostream>

// PRE: x is roundable to a number in the value range of type int
// POST: return value is the integer nearest to x, or the one further
//       away from 0 if x lies right in between two integers.
int round (double x) 
{
  int trunc = x; // rounds towards 0 by standard
  double error = x - trunc; // note: result is exact!

  if (error >= 0.5)
    // x was positive, and trunc + 1 is closer
    return trunc + 1;
  if (error <= -0.5)
    // x was negative, and trunc - 1 is closer
    return trunc - 1;

  // |error| < 0.5, trunc is closest integer
  return trunc;
}

int main ()
{
  // Input
  double x;
```
std::cin >> x;

// Output
std::cout << "The rounded number is " << round(x) << "\n";

return 0;