Introduction to Computer Science
Jacobs University Bremen
Dr. Jürgen Schönwälder

Course: CH-232-A
Date: 2021-10-01
Due: 2021-10-08

ICS 2021 Problem Sheet #4

Problem 4.1: *b-complement* (1+1+1 = 3 points)

We plan to use a fixed size b-complement number system with the base $b = 7$ and $n = 4$ digits.

a) What are the smallest and the largest numbers that can be represented and why?

b) What is the representation of -1 and -8 in b-complement notation?

c) Add the numbers -1 and -8 in b-complement notation. What is the result in b-complement representation? Convert the result from b-complement representation back into the decimal number system.

Solution:

a) We can represent $b^n = 7^4 = 2401$ different numbers. With 0 in the middle of the range, we can cover the range $[-1200, 1200]$:

```
-1200 -1199 ...   -2   -1    0    1    2     1199  1200 (decimal)
 3334  3335     6665 6666 0000 0001 0002 ... 3332  3333 (7-complement, 4 digits)
```

The range for a given base b and n digits can be calculated using this Haskell function:

```
1  range b n = (min, max)
2        where min = -(b^n `div` 2)
3              max =  (b^n `div` 2) - ((b+1) `mod` 2)
```

b) The absolute value of -1 in base 7 is 0001_7. We calculate for each digit $a'_i = (b-1) - a_i$, which gives us 6665. Adding 1 leads to the 4-digit b-complement base 7 representation 6666_{7b4}.

The absolute value of -8 in base 7 is 0011_7. We calculate for each digit $a'_i = (b-1) - a_i$, which gives us 6655. Adding 1 leads to the 4-digit b-complement base 7 representation 6656_{7b4}.

c) We add the numbers in 4-digit b-complement base 7 representation in the usual way (ignoring any carry over):

```
  6666
+ 6656
------
  6655
```

We calculate for each digit $a'_i = (b-1) - a_i$, which gives us 0011. Adding 1 leads to the base 7 representation 0012_7 of the absolute value, which is $7 + 2 = 9$ in the decimal number system. Hence, the overall result is -9.

Marking:

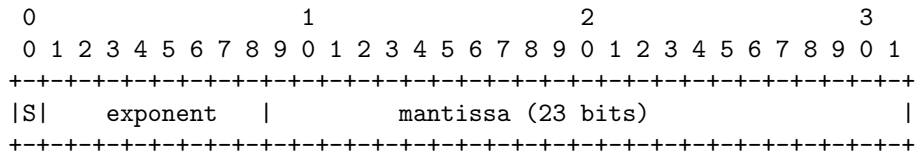
a) - *1pt for the correct range and reasoning*

b) - *0.5pt for each correct conversion into b-complement notation*

c) - *0.5pt for a correct addition of numbers in b-complement notation*
- *0.5pt for a correct conversion into decimal notation*

Problem 4.2: *IEEE 754 floating point numbers* (4+1 = 5 points)

IEEE 754 floating point numbers (single precision) use the following format (the numbers on the top of the box indicate bit positions, the fields in the box indicate what the various bits mean).



The encoding starts with a sign bit, followed by the biased exponent (8 bits), followed by the mantissa (23 bits). For single-precision floating-point numbers, the exponents in the range of -126 to +127 are biased by adding 127 to get a value in the range 1 to 254 (0 and 255 have special meanings).

a) 20.5 degree Celsius corresponds to 293.65 Kelvin. Explain step by step (and in your own words) how the decimal fraction 293.65 is converted into a single precision floating point number.

b) What is the decimal number that is actually stored in the single precision floating point number and what is the absolute error?

Solution:

a) Since the number 293.65_{10} is negative, we set the sign bit to 0.

In the next step, we convert the absolute integer part of 293.65_{10} into binary representation. We can use the dec2bin algorithm:

$$
\begin{aligned}
293 \bmod 2 &= 1 & 1_2 \\
146 \bmod 2 &= 0 & 01_2 \\
73 \bmod 2 &= 1 & 101_2 \\
36 \bmod 2 &= 0 & 0101_2 \\
18 \bmod 2 &= 0 & 00101_2 \\
9 \bmod 2 &= 1 & 100101_2 \\
4 \bmod 2 &= 0 & 0100101_2 \\
2 \bmod 2 &= 0 & 00100101_2 \\
1 \bmod 2 &= 1 & 100100101_2
\end{aligned}
$$

Next, we convert the fractional part of 293.65_{10} into a binary fraction using the decftobinf algorithm:

$0.65 \cdot 2 = 1.30$	1
$0.30 \cdot 2 = 0.60$	10
$0.60 \cdot 2 = 1.20$	101
$0.20 \cdot 2 = 0.40$	1010
$0.40 \cdot 2 = 0.80$	10100
$0.80 \cdot 2 = 1.60$	101001
$0.60 \cdot 2 = 1.20$	1010011
$0.20 \cdot 2 = 0.40$	10100110
$0.40 \cdot 2 = 0.80$	101001100
$0.80 \cdot 2 = 1.60$	1010011001
$0.60 \cdot 2 = 1.20$	10100110011
$0.20 \cdot 2 = 0.40$	101001100110
$0.40 \cdot 2 = 0.80$	1010011001100
$0.80 \cdot 2 = 1.60$	10100110011001
$0.60 \cdot 2 = 1.20$	101001100110011
$\ldots$	

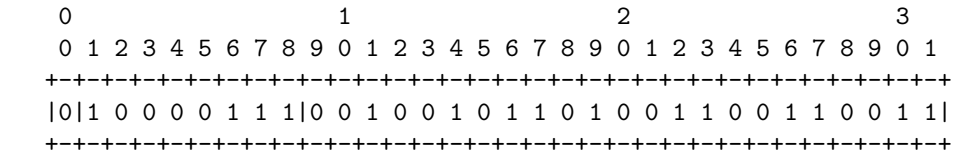
Combining the results of the last two steps, we obtain the truncated binary fraction

$$100100101.101001100110011_2$$

for the decimal value 293.65_{10}. Normalization of the binary fraction gives us:

$$1.00100101101001100110011_2 \cdot 2^8$$

Adding the bias 127_{10} to the exponent 8_{10}, we obtain the biased exponent $135_{10} = 10000111_2$. Putting things together and dropping the leading 1 of the mantissa, we obtain the following floating point number representation:



b) The decimal number encoded is:

$$1 \cdot (2^0 + 2^{-3} + 2^{-6} + 2^{-8} + 2^{-9} + 2^{-11} + 2^{-14} + 2^{-15} + 2^{-18} + 2^{-19} + 2^{-22} + 2^{-23}) \cdot 2^8$$
$$= 293.649993896484375$$

The error e and the absolute error $|e|$ are given by:

$$e = 293.649993896484375 - 293.65 = -0.000006103515625$$
$$|e| = 0.000006103515625$$

Marking:

a)
- *0.5pt for the sign bit*
- *0.5pt for the integer conversion to binary*
- *1pt for the fraction conversion to binary*
- *0.5pt for the normalization*
- *0.5pt for biasing the exponent*
- *0.5pt for converting the biased exponent to binary*
- *0.5pt for showing the resulting single precision floating point format*

b)
- *0.5pt for calculating the decimal fraction corresponding to the floating point value*
- *0.5pt for calculating the absolute error*

Problem 4.3: *munged passwords (haskell)* (1+1 = 2 points)

Some people try to create stronger passwords through character substitution. The substitutions can be anything the user finds easy to remember. We use the following substitution:

character	a	b	c	d	e	f	g	h	i	l	o	q	s	x	y
substitution	@	8	(	6	3	#	9	#	1	1	0	9	$	%	?

Using this table, the string `hello world` is munged into the string `#3110 w0r16`.

a) Using pattern matching, implement a function `sub` that takes a character and returns either the character or a substitution of it. Write down the type signature of your function followed by its definition.

b) Using pattern matching, implement a function `munge` that takes a string and returns a string with all character substitutions applied. Write down the type signature of your function followed by its definition.

Submit your Haskell code plus an explanation (in Haskell comments) as a plain text file.

Solution:

a) A possible solution (not requiring language features not introduced yet):

```haskell
sub :: Char -> Char
sub 'a' = '@'
sub 'b' = '8'
sub 'c' = 'C'
sub 'd' = '6'
sub 'e' = '3'
sub 'f' = '#'
sub 'g' = '9'
sub 'h' = '#'
sub 'i' = '1'
sub 'l' = '1'
sub 'o' = '0'
sub 'q' = '9'
sub 's' = '$'
sub 'x' = '%'
sub 'y' = '?'
sub c = c
```

A somewhat smarter solution:

```haskell
sub :: Char -> Char
sub c = subs c smap
    where smap = "a@b8c(d6e3f#g9h#i1l1o0q9s$x%y?"
          subs c [] = c
          subs c [_] = c
          subs c (a:b:xs) = if a == c then b else subs c xs
```

One could have used a list of tuples but then writing down the list of tuples becomes pretty verbose.

b) A simple recursion:

```haskell
munge :: [Char] -> [Char]
munge [] = []
munge (x:xs) = sub x : munge xs
```

Of course, this is just a special case of a map (and we can even write it in a point-free notation):

```haskell
munge :: [Char] -> [Char]
munge = map sub
```

Marking:

a)
- *0.5pt for the type signature*
- *0.5pt for the function definition*

b)
- *0.5pt for the type signature*
- *0.5pt for the function definition*