Problem Sheet 5

Henri Sota
h.sota@jacobs-university.de
Computer Science 2022

October 18, 2019

Problem 5.1

Consider a b-complement number system with the base $b = 5$ and $n = 4$ digits.

a) In order to represent -1 and -8 in b-complement notation, first we need to find the representation of the absolute value of each number in base b.

$$|-1_{10}| = 1_{10} = 0001_5 \qquad |-8_{10}| = 8_{10} = 0013_5$$

Next, we need to find the complement of our numbers in base b using $a'_i = (b-1) - a_i$:

$$0001_5 \rightarrow 4443_5 \qquad 0013_5 \rightarrow 4431_5$$

We add 1 to each number as the last step:

$$4443_5 + 1_5 = 4444_5 \qquad 4431_5 + 1_5 = 4432_5$$

So, b-complement notation of -1 and -8 in base 5 with 4 digit representation are respectively 4444_5 and 4432_5.

b) Addition in b-complement notation of -1 and -8:

$$\begin{array}{r} 4\,4\,4\,4 \\ +\ 4\,4\,3\,2 \\ \hline (1)\,4\,4\,3\,1 \end{array}$$

Most significant digit is omitted (1) and the number that we got as an answer is a negative number. In order to convert back to decimal, we invert the bits using $a_i = a'_i - b + 1$, we add 1 to the result and convert to decimal.

$$4431_5 \rightarrow 0013_5$$

$$0013_5 + 1_5 = 0014_5 \rightarrow 9_{10}$$

That is the absolute value of the negative number -9 which is the sum of -1 and -8.

Problem 5.2

In order to convert a decimal fraction to a single precision floating point number, we take the following steps:

1. Sign bit S value is set by the sign of the decimal fraction. If the sign is positive, $S = 0$, else if the sign is negative, $S = 1$.
2. Convert the absolute value of our number to binary by dividing the decimal fraction into the integral part and the fraction part and then combining them together. The integral part is converted using the division and remainder method, while the fraction part is converted using multiplication by 2 and subtraction by 1 in case multiplication produces a number bigger or equal to 1.
3. Normalize the number such that the integral part I is $1 \geq I \geq 10$ and the exponent of 2 will be the amount of digits which we took from integral part or the negative of the amount of digits we gave to make the integral part bigger than 0.
4. Add the bias, which is equal to 2 raised to the power of the number of bits used to represent the exponent part in our single precision floating point number minus 1, minus 1: $2^{t-1} - 1$ (where t is the number of bits).
   In our case, bias is equal to 127. $(2^{8-1} - 1 = 127)$
5. Mantissa is the binary representation of the integral and fractional part, omitting the leading digit. (adding 0s at the end of our mantissa in case calculations produce less than 23 binary digits)

a) Sign bit S for -273.15_{10} is 1, because decimal fraction is negative.
Converting integral part and fraction part separately:

$$\begin{aligned} 273_{10} &= 100010001_2 \\ 0.15_{10} &= 0010011001100110011_2 \end{aligned}$$

(Multiplication and subtraction by 1 method on 0.15_{10} gives: $0.3(0) \to 0.6(0) \to 1.2(1) \to 0.4(0) \to 0.8(0) \to 1.6(1) \to 1.2(1) \ldots \infty$ which produces a binary representation $0010011001\ldots_2$ as the part $''1001''$ keeps on repeating.) Our binary representation after combination becomes:

$$100010001.0010011001100110011_2$$

After normalization:

$$1.0001000100100110011001100110011_2 \cdot 2^8$$

Calculating exponent with bias:

$$8_{10} + 127_{10} = 135_{10} = 10000111_2$$

Single precision floating point number representation of -273.15 is:

$$\underbrace{1}_{\text{S}}\underbrace{10000111}_{\text{exponent}}\underbrace{00010001001001100110011}_{\text{mantissa}}{}_2$$

b) The decimal fraction actually stored in the single precision floating point number is less than -273.15, -273.149993896484375. To calculate the decimal fraction, the first bit is the sign bit, 8 following bits are the exponent biased by 127 and the last 23 bits are converted using the formula: $\text{value}(i) * 2^{-i}$ for $i = 1, 2, 3, \ldots$. Bringing it all up, the result is calculated using: $(-1)^S \cdot (1 + m) \cdot 2^e$ where S is sign bit, e is exponent unbiased and m is mantissa converted to decimal.
Calculation: $(-1)^1 \cdot (2^8) \cdot (1 + (2^{-4} + 2^{-8} + 2^{-11} + 2^{-14} + 2^{-15} + 2^{-18} + 2^{-19} + 2^{-22} + 2^{-23}))$

Problem 5.3

Functions implemented to help with the solutions:

```haskell
-- Imported function digitToInt to convert from Char to Int
import Data.Char(digitToInt)

-- Function takes 2 Ints, number and base, and returns a string
-- Functions works for numbers bigger than 9 that need to be represented by
-- A,B,C,D,E,F for bases bigger than 10
-- It converts 10->A, 11->B, ...
convertOver10 :: Int -> Int -> String
convertOver10 number base
  | remainder == 10 = "A"
  | remainder == 11 = "B"
  | remainder == 12 = "C"
  | remainder == 13 = "D"
  | remainder == 14 = "E"
  | remainder == 15 = "F"
  | otherwise = show remainder
  where remainder = mod number base

-- Function takes a Char and returns an Int
-- It performs the opposite of convertOver10
-- It reverts back A->10, B->11, ...
revertOver10 :: Char -> Int
revertOver10 number
  | number == 'A' = 10
  | number == 'B' = 11
  | number == 'C' = 12
  | number == 'D' = 13
  | number == 'E' = 14
  | number == 'F' = 15
  | otherwise = (digitToInt number :: Int)

-- Function takes 2 Ints, base and counter, and a String, representation of a
-- number in that base, and it returns an Int, the decimal equivalent of the
-- number by recursively going from the end to the beginning of the String
-- and summing up the values of each digit in String
sumDigits :: Int -> Int -> String -> Int
sumDigits _ _ [] = 0
sumDigits base i lst = ((base ^ i) * (revertOver10 (last lst)) + (sumDigits base (
    i+1) (init lst)))
```

a)
```haskell
-- a) Code toBase
-- Function takes 2 Ints, base and number, and returns a list of Ints
-- which is the representation of number in the base
-- It works by recursively calling itself and appending the remainder of our
-- number divided by our base at the end of the list
toBase :: Int -> Int -> [Int]
toBase base number
  | base < 2    = []
  | number <= 0 = []
  | otherwise = (toBase base (div number base)) ++ [mod number base]
```

b)
```haskell
-- b) Code fromBase
-- Function takes 1 Int, base, 1 list of Ints, representation of a number in
-- that base, and it returns an Int, the decimal equivalent of the number
-- It works its way from the end of the list and it sums up the values of
-- each digit in list
fromBase :: Int -> [Int] -> Int
fromBase base lst
  | base < 2         = 0
  | null lst         = 0
  | all (== 0) lst   = 0
  | otherwise = sumDigit 0 (reverse lst)
  where
    sumDigit i [] = 0
    sumDigit i (x:xs)
      | x < 0 = sumDigit  i []
      | otherwise = ((base ^ i) * x) + sumDigit (i + 1) xs
```

c)
```
-- c) Code showBase
-- showBin - convenience function to represent a number in binary
-- It appends to the end of the string the remainder of division between
-- number and 2 and it calls itself recursively with argument the
-- integer division between number and 2
showBin :: Int -> String
showBin 0 = "0"
showBin 1 = "1"
showBin number
  | mod number 2 == 0 = showBin (div number 2) ++ "0"
  | otherwise = showBin (div number 2) ++ "1"

-- showOct - convenience function to represent a number in octal
-- It appends to the end of the string the remainder of division between
-- number and 8 and it calls itself recursively with argument the
-- integer division between number and 8
showOct :: Int -> String
showOct number
  | number < 8 = show number
  | otherwise = (showOct (div number 8)) ++ (show (mod number 8))

-- showHex - convenience function to represent a number in octal
-- It appends to the end of the string the remainder of division between
-- number and 16 and it calls itself recursively with argument the
-- integer division between number and 16
-- In case that remainder is between 9 and 16, it appends the respective
-- character in hexadecimal by calling convertOver10
showHex :: Int -> String
showHex 0 = ""
showHex number
  | ((mod number 16) < 16) && ((mod number 16) > 9 ) = (showHex (div number
    16) ++ (convertOver10 number 16))
  | number == 0 = ""
  | otherwise = showHex (div number 16) ++ (show (mod number 16))

-- Function takes 2 Ints, base and number, and returns String
-- It converts the decimal number into a number in the base that we got as arg
-- It works by appending to the end of the string, the remainder of division
-- between number and base and it calls itself
-- It calls itself recursively by performing integer division and using the
-- remainder of the division between number and base to produce the digits
-- that represent the number in that base
-- It utilizes convenience functions showBin, showOct and showHex when
-- base is 2, 8 or 16 respectively
showBase :: Int -> Int -> String
showBase base number
  | number < 0    = ("Error: Number is negative")
  | number == 0   = ""
  | base < 2      = ("Error: Base less than 2")
  | base == 2     = showBin number
  | base == 8     = showOct number
  | base == 16    = showHex number
  | base > 10     = (showBase base (div number base)) ++ (convertOver10 number
     base)
  | otherwise     = (showBase base (div number base)) ++ (show (mod number
    base))
```

d)
```
-- d) Code readBin
-- readBin - convenience function to calculate decimal representation
-- by calling sumDigits with argument of base as 2
readBin :: String -> Int
readBin lst = sumDigits 2 0 lst

-- readOct - convenience function to calculate decimal representation
-- by calling sumDigits with argument of base as 8
readOct :: String -> Int
readOct lst = sumDigits 8 0 lst

-- readHex - convenience function to calculate decimal representation
-- by calling sumDigits with argument of base as 16
readHex :: String -> Int
readHex lst = sumDigits 16 0 lst

-- Function takes an Int, base, a String, representation of a number in
-- that base, and returns an Int, the decimal equivalent of that number
-- It calculates the decimal value by calling function sumDigits
-- It utilizes convenience functions readBin, readOct and readHex when
-- base is 2, 8 or 16 respectively
readBase :: Int -> String -> Int
readBase base lst
  | base < 2          = 0
  | null lst          = 0
  | all (== '0') lst  = 0
  | base == 2         = readBin lst
  | base == 8         = readOct lst
  | base == 16        = readHex lst
  | otherwise = (sumDigits base 0 lst)
```

In order to test these functions, I've used multiple calls to these with different inputs, by mapping the different inputs coming from a list comprehension with each function:

```
print(map (\b -> toBase b 16) [2..12])
print(map (\b -> fromBase b [0,1,1]) [2..16])
print(map (\b -> showBase b 16) [2..16])
print(map (\b -> readBase b "111") [2..16])
```

which produced:

```
[[1,0,0,0,0],[1,2,1],[1,0,0],[3,1],[2,4],[2,2],[2,0],[1,7],[1,6],[1,5],[1,4]]
[7,13,21,31,43,57,73,91,111,133,157,183,211,241,273]
["10000","121","100","31","24","22","20","17","16","15","14","13","12","11","10"]
[7,13,21,31,43,57,73,91,111,133,157,183,211,241,273]
```

Also I've tested functions showBase and readBase if they were able to output and read cases when base was bigger than 10:

```
print(map (\b -> showBase 16 b) [2..16])
print(map (\b -> readBase 16 b) ["1","F","2F","A3C"])
```

which produced:

```
["2","3","4","5","6","7","8","9","A","B","C","D","E","F","10"]
[1,15,47,2620]
```