Problem Sheet 6

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

October 23, 2019

Problem 6.1

a) Converting each hexadecimal value to the binary system:
83 105 109 112 108 105 99 105 116 121 44 32 99 97 114 114
105 101 100 32 116 111 32 116 104 101 32 101 120 116 114 101
109 101 44 32 98 101 99 111 109 101 115 32 101 108 101 103
97 110 99 101 46 10 45 32 74 111 110 32 70 114 97 110
107 108 105 110 10

Converting each decimal value to its respective character on the ASCII table:
Simplicity, carried to the extreme, becomes elegance.
- Jon Franklin

b) Table containing information about the corresponding Unicode code point and character name of several UTF-8 encoded characters represented in hexadecimal

UTF-8	**Unicode**	**Name**
0a	U+000A	LINE FEED (LF)
20	U+0020	SPACE
2d	U+002D	HYPHEN-MINUS
37	U+0037	DIGIT SEVEN
3d	U+003D	EQUALS SIGN
7b	U+007B	LEFT CURLY BRACKET
7c	U+007C	VERTICAL LINE
7d	U+007D	RIGHT CURLY BRACKET
c2 ad	U+00AD	SOFT HYPHEN
e2 80 91	U+2011	NON-BREAKING HYPHEN
e2 80 92	U+2012	FIGURE DASH
e2 80 93	U+2013	EN DASH
e2 80 94	U+2014	EM DASH
e2 88 92	U+2212	MINUS SIGN
e2 88 aa	U+222A	UNION

c) UTF-8 is a multibyte encoding able to encode the whole Unicode charset. An encoded character takes between 1 and 4 bytes. In our case the hexadecimal representation of each character in the CJK Unified Ideographs code block utilizes 3 bytes or 24 bits. In order to represent a Chinese text made up of 800000 characters, we would need:
3 bytes per character $\cdot$ 800000 characters $=$ 2400000 bytes in total with UTF-8.

UTF-32 takes 32 bits or 4 bytes to represent each character in the Unicode charset. In order to represent a Chinese text made up of 800000 characters, we would need:
4 bytes per character $\cdot$ 800000 characters $=$ 3200000 bytes in total with UTF-32.

Problem 6.2

a) **No.3** and **No.5** are equivalent with each other. Converting **No.5** to the same time zone as **No.3** $(+00:00)$ by subtracting the time zone from the hour:
$00:30:00-(-12:45:00) \rightarrow 13:15:00$
No.2, **No.4** and **No.6** are equivalent with each other. Converting **No.4** and **No.6** to the same time zone as **No.2** $(+00:00)$ by subtracting the corresponding time zone from the hour respectively:
No.4: $15:15:00-(-02:00:00) \rightarrow 17:15:00$
No.6: $05:15:00-(+12:00:00) \rightarrow 17:15:00$ (of the "previous" day)
No. 1 is not equivalent to any other date as converted to UTC $+00:00$ is
$2019-10-15T13:15:00+00:00$

b) RFC 3339 is listed as a profile of ISO 8601, with a few little differences regarding 4-digit years, standard format, representation of fractional seconds. The time zone offset $+00:00$ indicates a time zone at Universal Time. By definition of RFC3339, if the time in UTC is known, but the offset to local time is unknown, this can represented with an offset of $-00:00$.

Another interesting convention regarding the $-00:00$ offset is also RFC 2822 about email time stamps.
RFC 2822: Although a time zone offset of $-00:00$, indicates that the time was generated on a system that may be in a local time zone other than Universal Time. Therefore it indicates that the date-time contains no information about the local time zone.

c) Year 2038 Problem is a problem regarding the time data type representation on 32-bit processor architectures. On 32-bit systems, time is saved on a 32-bit integer. It takes values from 0 to 2147483647. Therefore the amount of seconds time can count up to is 2147483647 seconds. At 03:14:07 UTC on January 19, 2038, the number of seconds since January 1, 1970.
Solution: One solution provided to solve this problem is to upgrade systems to 64-bit processors which can count up to 9^{18}, so an integer overflow won't happen on these systems. The integer overflow would make these 32-bit architectures to return to the same date of January 1, 1970.

Problem 6.3

a)
```
-- a) Code enc
-- Function takes a String and returns a String
-- It recursively goes through the String and checks if the head of the String
-- is an upper or lower case character
-- If so, it converts them to emojis
-- Else it leaves the character as it is and calls itself with the String tail
enc :: String -> String
enc [] = ""
enc (x:xs)
    | isLower x = [(toEnum) ((fromEnum x) + 128415)] ++ enc xs
    | isUpper x = [(toEnum) ((fromEnum x) + 127935)] ++ enc xs
    | otherwise = [x] ++ enc xs
```

```
-- b) Code dec
-- Function takes a String and returns a String
-- It recursively goes through the String and checks if the decimal value of
-- is between a certain interval of 26 numbers after 128512 or 128000
-- If so, it converts them back to their respective alphabetic character
-- Else it leaves the character as it is and calls itself with the String tail
dec :: String -> String
dec [] = ""
dec (x:xs)
    | ((fromEnum x) >= 128512 && (fromEnum x) <= 128537) = [(toEnum) ((
   fromEnum x) - 128415)] ++ dec xs
    | ((fromEnum x) >= 128000 && (fromEnum x) <= 128025) = [(toEnum) ((
   fromEnum x) - 127935)] ++ dec xs
    | otherwise = [x] ++ dec xs
```

The first character of the lowercase group of characters $'a'$ has code point $U + 0061$ (dec: 97) and the first character of the uppercase group of characters $'A'$ has code point $U + 0041$ (dec: 65). The first character of the emoji group of characters 'GRINNING FACE' has code point $U + 1F600$ (dec: 128512) and the first character of the animal group of characters 'RAT' has code point $U + 1F400$ (dec: 128000).
The difference between the first character of the lowercase group and the first character of the emoji group is 128415. The difference between the first character of the uppercase group and the first character of the animal group is 127935.

- Code was compiled on an online compiler tio.run as ghci consideres emoji characters as halfwidth while the terminal does correctly treat them as fullwidth.