2020 Haskell January Test
Supplementary notes

1 Bit twiddling

This year's Haskell test will involve some bit manipulation, which is sometimes referred to as "bit twiddling". To make this easy you'll be invited to use some of the functions from the Haskell module `Data.Bits`. The objective of these notes is to give you a brief introduction to the functions in `Data.Bits` and explain how to use them to implement some common bit twiddling operations.

Module `Data.Bits`

The `Data.Bits` module contains a type class which starts off like this:

```
class Eq a => Bits a where ...
```

For each type that is an instance of `Bits` the idea is to be able to manipulate objects of that type as *bit vectors*. The only instance of interest to the test is that for `Int`. The version of GHC you will be using supports 64-bit integers, so every object of type `Int` can be thought of as being a vector of 64 bits. For example, the integer 29855 has the following bit vector (binary) representation:

```
0000000000000000000000000000000000000000000000000111010010011111
```

Wow - that's a lot of bits! To keep things a little easier to write out in these notes we'll use examples containing "small" integers so we only have to focus on the least significant n bits for some "small" n. For example, we can display 29855 using as few as 15 bits, as it's most significant bit is at index 14 (note that bit indices start at 0). Here it is written out with 16 bits:

```
0111010010011111
```

The remaining 48 bits are there, of course, and can be used if necessary, so you shouldn't forget them.

To help you to practice with `Data.Bits` a file `practice.hs` has been uploaded to the Materials page. This imports `Data.Bits` and implements a useful function, `showBitVector :: Int -> Int -> String`, for displaying bit vector representations of integers. The first argument is the integer (bit vector) to display and the second specifies the number of bits to generate in the resulting string. For example:

```
*Main> showBitVector 29855 16
"0111010010011111"
```

The member functions of `Data.Bits` include the following, which we'll explain with examples. Each of these is implemented directly in low-level machine code instructions, so are super-fast. Somewhat unusually, we'll be quite interested in performance in this year's test.

Bit-wise “and” and “or”

The operators `(.&.) :: a -> a -> a` and `(.|.)  ::  a -> a -> a` respectively implement bit-wise “and” and “or” on the bit vector representation of objects of type `a`. Here, we’re only interested in the `Int` instance where the operators have the types:

```
(.&.) :: Int -> Int -> Int
(.|.) :: Int -> Int -> Int
```

We’ll apply a similar instantiation when describing the other member functions below.

The expression `i .&. j` performs a boolean (2-way) “and” on each bit of `i` and `j` pair-wise. The “and” works similarly to Haskell’s built-in `&&` operator, except that it works on bits (0 and 1) rather than booleans (`False` and `True`). Similarly, `i` .. j— applies “or” pair-wise, where “or” works similarly to `||`. For example:

```
*Main> showBitVector 9 4
"1001"
*Main> showBitVector 5 4
"0101"
*Main> 9 .&. 5
1
*Main> showBitVector (9 .&. 5) 4
"0001"
*Main> 9 .|. 5
13
*Main> showBitVector (9 .|. 5) 4
"1101"
```

`shiftL` and `shiftR`

The functions `shiftL ::  Int -> Int -> Int` and `shiftR ::  Int -> Int -> Int` shift all the bits of a bit vector left/ right respectively. The first argument is the bit vector (integer) and the second is the number of positions to shift. Zeros are added at the right/left bit positions accordingly. For example:

```
*Main> showBitVector 2000 16
"0000011111010000"
*Main> showBitVector (shiftL 2000 2) 16
"0001111101000000"
*Main> shiftL 2000 2
8000
*Main> showBitVector (shiftR 2000 3) 16
"0000000011111010"
*Main> shiftR 2000 3
250
```

Remark: Although it’s not important, you might note that shifting left/right by n positions is equivalent to multiplying/dividing by 2^n using integer arithmetic.

`bit`

The function `bit ::  Int -> Int` generates an integer whose bit vector representation has a 1 at the specified bit index (`Int`) and 0s elsewhere. The least significant bit has index 0. For example:

```
*Main> bit 0
1
*Main> showBitVector (bit 0) 8
"00000001"
*Main> bit 5
32
*Main> showBitVector (bit 5) 8
"00100000"
```

Note that `bit n` is equivalent to $2^{\mathtt{n}}$.

`setBit` and `clearBit`

The functions `setBit :: Int -> Int -> Int` and `clearBit :: Int -> Int -> Int` respectively set and clear the bit at the specified index (second argument) in a given bit vector (first argument). For example:

```
*Main> showBitVector 39855 16
"1001101110101111"
*Main> setBit 39855 14
56239
*Main> showBitVector (setBit 39855 14) 16
"1101101110101111"
*Main> clearBit 39855 0
39854
*Main> showBitVector (clearBit 39855 0) 16
"1001101110101110"
```

`testBit`

The function `testBit :: Int -> Int -> Bool` returns `True` if the bit at the specified index (second argument) in a given bit vector (first argument) is 1; `False` otherwise. For example:

```
*Main> showBitVector 1771 12
"011011101011"
*Main> testBit 1771 3
True
*Main> testBit 1771 8
False
```

1.1 Masking

A common operation on bit vectors is *masking*, which involves using `.&.`, often in conjunction with `bit` and `shiftL/R`, to zero out all but a specified set of bits in a bit vector. First, it's useful to note that `bit n - 1` $= 2^{\mathtt{n}} - 1$ is a bit vector with 1s in positions 0...`n-1` and 0s elsewhere; it's often referred to as a *mask* because "anding" it with a given bit vector has the effect of zeroing, i.e. masking out, all bits of the bit vector where the mask is 0. For example:

```
*Main> showBitVector (bit 6 - 1) 16
"0000000000111111"
*Main> showBitVector 44628 16
```

```
"1010111001010100"
*Main> showBitVector (44628 .&. (bit 6 - 1)) 16
"0000000000010100"
```

This gives you the least significant 6 bits of 44628. Any other contiguous set of bits can be extracted by applying `shiftR`, for example, the expression `shiftR 44628 5 .&. (bit 6 - 1)` extracts bits 5-10 (underlined) of $44628{=}10101\underline{110010}10100_2$, returning the result $50 = 110010_2$ as an integer:

```
*Main> showBitVector 44628 16
"1010111001010100"
*Main> showBitVector (shiftR 44628 5) 16
"0000010101110010"
*Main> showBitVector (bit 6 - 1) 16
"0000000000111111"
*Main> shiftR 44628 5 .&. (bit 6 - 1)
50
*Main> showBitVector (shiftR 44628 5 .&. (bit 6 - 1)) 16
"0000000000110010"
```

That's all you need for now. Get practising! These notes will be made available to you during the test.