1. The following program describes poems as a sequence of lines, or verses. You may assume that the code for the functions whose definition is not shown is, in fact, given.

```haskell
-- Split a verse into separate words
toWords :: Verse -> [Word]
toWords ...

-- Find the last word in a verse
lastWord :: Verse -> Word
lastWord ...

-- Count the number of syllables in a word
countSyllablesInWord :: Word -> Integer
countSyllablesInWord ...
```

Answer question 1 and any other question. Question 1 is worth 40 marks, while questions 2 and 3 carry 60 marks each. If more than one additional question is answered, only the first answer in the booklet will be taken into consideration. Students are allowed to use course notes, books and calculators.
-- Check whether two words rhyme
rhyme :: Word -> Word -> Bool
rhyme ...

Show your understanding of the above program by giving the type and defining the function `countWordsInVerse`, which given a `Verse`, returns the number of words in the verse. Similarly define a function `countSyllablesInVerse` which counts the number of syllables in a verse.

Two verses rhyme if their last word rhyme. Define `versesRhyme`, which given two verses, returns whether they rhyme.

A limerick is a five line poem, in which (i) the first, second and fifth verses rhyme; (ii) the third and fourth verses rhyme; (iii) the first, second and fifth verse are each seven or eight syllables long (they may be of equal or different length); and (iv) the third and fourth verse are five or six syllables long (they may be of equal or different length). Define the function `isLimerick`, which tells you whether or not a given poem is a limerick.

What do the following three functions do? What are their types?

```haskell
question1 = head . reverse . toWords

question2 xs =
  [ show n ++ ":" ++ x | (n, x) <- zip [1..] xs ]

question3 (x:xs) = and (map (rhyme x) xs)
  where
  and [] = True
  and (x:xs) = x && and xs
```

2. Consider the following datatype definition:\footnote{Please note that this type and the related functions are intended solely for theoretical purposes. Do not try this out at home unless in the presence of an adult.}:
data Drink =
    Beer
| Vodka
| Double Drink
| Mix Drink Drink

The datatype can be used to describe cocktails. The basic drinks are Beer and Vodka representing one pint of beer and vodka respectively. The Double of a drink can be seen as making two copies of the drink and mixing them together. For example Double (Double Vodka) would contain four pints of vodka. Finally, Mix of two drinks gives a mixture of the drinks.

Define functions vodkaContent and beerContent, which return the number of pints of vodka and beer in a given drink respectively.

Two drinks are considered equal if they contain the same amount of beer and the same amount of vodka. Make Drink an instance of the typeclass Eq following this definition.

Make a new typeclass Addition, such that instances a of the class support addition (add :: a -> a -> a) and doubling (dbl :: a -> a). Make the default implementation of doubling to be the action of adding the item to itself.

Make Int and Drink instances of Addition, implementing both functions. Strings can also be made an instance of Addition, by interpreting addition as ++. Implement it.

What are the types of the following functions?

```haskell
question1 = dbl . dbl

question2 x y =
    if x == y then x else dbl y

question3 x y z
    | dbl x == y = z
    | otherwise = dbl z
```
3. Consider the following datatype definition:

```haskell
data SmallCollection a =
  Empty
  | One a
  | Two a a
```

The datatype can be used to describe a small collection of up to two objects of a particular type `a`. For instance, `SmallCollection Int` will contain either nothing (Empty), one integer (eg One 6) or two integers (eg Two 4 7). In a small collection, one may have two identical objects eg Two 3 3.

Define a function `containsMore`, which given two small collections of the same type, returns whether the first contains more items than the second. Specify the type of `containsMore`.

Define a function `isContainedIn`, which given two small collections, returns whether every element of the first is contained in the second (including repetitions). For example, `Empty `isContainedIn` Two 1 4` should be true, like `One 4 `isContainedIn` Two 1 4` and also like `Two 4 1 `isContainedIn` Two 1 4`. On the other hand the expression `Two 1 4 `isContainedIn` One 4` should be false, just like the expression `One 3 `isContainedIn` Two 1 4` and also the expression `Two 1 1 `isContainedIn` Two 1 4`. Give the type of `isContainedIn`.

Make `SmallCollection a` an instance of the `Eq` typeclass. Two small collections should be equal to each other if every element of one is in the other and vice-versa.

Specify the types of the following functions:

```haskell
question1 Empty = False
question1 (One x) = False
question1 (Two x y) = x==y

question2 _ [] = []
question2 f (Empty:xs) = question2 f xs
question2 f (One x:xs) = f x: question2 xs
question2 f (Two x y:xs) = f x: f (f y): question2 f xs
```
question3 [] = []
question3 (Empty:xs) = question3 xs
question3 (One x:xs) = One (x 'a'): question3 xs
question3 (Two x y:xs) = Two (x 'a') (y 'a'): question3 xs

question4 f = Two f (f.f)