

Except where otherwise stated, any code you write in this section should be in Haskell.

Question 1

- (a) Consider an operation `scan` which computes the prefix sum on lists of arbitrarily large integers. When given a list $[x_0, x_1, x_2, \dots, x_{n-1}]$, `scan` should return the list $[y_0, y_1, y_2, \dots, y_{n-1}]$ where $y_0 = x_0$, $y_1 = x_0 + x_1$, and generally $y_j = \sum_{i=0}^j x_i$.

- i. Implement `scan` recursively. Make use of pattern matching in your answer. **[10 Marks]**

- (b) Now turn `scan` into a polymorphic, higher-order function

- i. Rewrite `scan` as a new function, `scanf`, which accepts an additional argument that can be any binary operator. Ensure that `scanf` continues to yield the results of `scan` if you pass in `(+)` as the higher-order argument. **[4 Marks]**

- (c) Haskell uses **lazy evaluation** rules. Describe how this differs from **eager evaluation** (as seen in C# or C) with reference to functions and their arguments. **[4 Marks]**

- (d) The Haskell evaluator treats expressions as graphs which contain a combination **reducible expressions** (or redexes) and **irreducible expressions**. Two particularly important types of expression graphs are **normal form** and **weak head normal form (WHNF)**. Draw the corresponding expression graph for each of the following Haskell expressions, and state if they are in **normal form**, **WHNF**, or neither.

- i. `1 + 2` **[2 Marks]**

- ii. `1 : 2 : []` **[2 Marks]**

- iii. `fact = 1 : zipWith (*) [1..] fact` **[4 Marks]**

- (e) Outermost evaluation can have drawbacks for code performance, and so Haskell provides the `$!` operator for strict evaluation. Explain briefly why outermost evaluation might exhibit bad performance, and how strict evaluation can help. **[4 Marks]**

Question 2

- (a) Functional languages are increasingly being used for parallel and distributed computing. A prototypical example is the `mapReduce` framework, which provides **automatic** parallelism. That is, it can automatically determine parts of a computation which may safely execute in parallel.

In addition to this use case, optimising compilers often use a functional-inspired intermediate representation when they are implementing code transformations such as determining if function calls can be inlined or exchanged for optimised versions.

Discuss why you think that the functional paradigm is being adopted in these spheres. You could consider

- how easy it is to deduce which code is safe to execute in parallel;
- when program transformations are safe to apply;
- verifying correctness of transformations.

You may also consider other points. You may wish to illustrate your arguments with pseudocode. **[20 Marks]**

Section A Object oriented programming
(Dr Hubert Shum)

Except where otherwise stated, the questions in this Section should be answered assuming a C# environment using the Microsoft .NET Framework Class Library.

Question 3

- (a) Explain the key advantage of using a public property to access a private class variable, instead of a public variable, with no more than 50 words.
[4 Marks]
- (b) Explain three major differences between an abstract class and an interface, with no more than 30 words for each difference.
[6 Marks]
- (c) Explain the major similarity and the major difference between method hiding and method overriding, with no more than 70 words.
[6 Marks]
- (d) With the help of an example, explain how graphical user interface programming with Windows Presentation Foundation (WPF) utilises the inheritance concept of object-oriented programming, with no more than 50 words.
[4 Marks]

Question 4

You are asked to develop a `NumberList` class for some list operations. The partial source code of the `NumberList` class, a main program demonstrating the functionalities and the expected outputs are shown below. You are asked to implement the functionalities that can generate the outputs as shown. You should also do proper checking to ensure valid operations and memory storage.

The NumberList Class

```
public class NumberList
{
    private int iSize;
    private List<int> lstI = new List<int>();
    private Random ran = new Random();

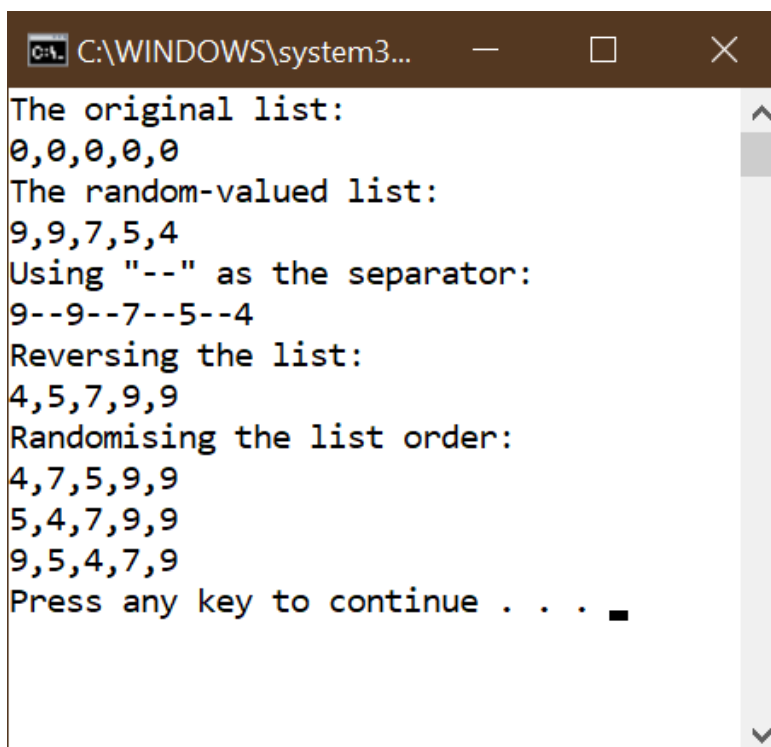
    public NumberList(int iSize)
    {
        this.iSize = iSize;
        for (int i = 0; i < iSize; i++)
            lstI.Add(0);
    }

    // Your implementations start here

}
```

The Main Program

```
static void Main(string[] args)
{
    NumberList N = new NumberList(5);
    Console.WriteLine("The original list:");
    N.Print(",");
    N.RandomValues();
    Console.WriteLine("The random-valued list:");
    N.Print(",");
    Console.WriteLine("Using \"--\" as the separator:");
    N.Print("--");
    Console.WriteLine("Reversing the list:");
    N.Reverse();
    N.Print(",");
    Console.WriteLine("Randomising the list order:");
    N.RandomOrder();
    N.Print(",");
    N.RandomOrder();
    N.Print(",");
    N.RandomOrder();
    N.Print(",");
}
```



```
C:\WINDOWS\system32\cmd.exe
The original list:
0,0,0,0,0
The random-valued list:
9,9,7,5,4
Using "--" as the separator:
9--9--7--5--4
Reversing the list:
4,5,7,9,9
Randomising the list order:
4,7,5,9,9
5,4,7,9,9
9,5,4,7,9
Press any key to continue . . .
```

The output of the main program.

continued

- (a) Write the function `RandomValues` in the `NumberList` class, which updates the values of each item in `lstI` with a random integer between 1 and 9 (inclusive of 1 and 9). Then, explain the programming logic with no more than 30 words. **[4 Marks]**
- (b) Write the function `Print` in the `NumberList` class, which prints the content of `lstI`, with a separator text between two numbers passed as a function parameter. Then, explain the programming logic with no more than 50 words. **[6 Marks]**
- (c) Write the function `Reverse` in the `NumberList` class, which reverses the order of the items in `lstI`. Then, explain the programming logic with no more than 50 words. **[8 Marks]**
- (d) Write the function `RandomOrder` in the `NumberList` class, which randomises the order of the items in `lstI`. Then, explain the programming logic with no more than 70 words. **[12 Marks]**