

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]**

Solution: *Knowledge, Application*

```
scan :: [Integer] -> [Integer]
scan [] = []
scan [x] = [x]
scan (x:y:xs) = x : scan (x+y:xs)
```

- `scan :: [Integer] -> [Integer]` [2 marks]. Only [1 mark] if `Int`.
- `scan [] = []` [1 mark]
- `scan [x] = [x]` [1 mark]
- `scan (x:y:xs)` [2 marks] for the pattern match `x:y:xs`, [1 mark] for the brackets.
- `= x : scan (x+y:xs)` [1 mark] for the concatenation, [1 mark] for the recursive call, [1 mark] for application to `(x+y:xs)`.

- (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]**

Solution: *Comprehension, Application*

```
scanf :: (a -> a -> a) -> [a] -> [a]
```

```
scanf _ [] = []
scanf _ [x] = [x]
scanf f (x:y:xs) = x : scanf f (x 'f' y : xs)
```

- Function argument is binary (three arguments) [1 mark]
- Generic type variable *a* for all arguments [1 mark]
- Extra parameter *f* (or similar) [1 mark]
- Replacing *x+y* with *x 'f' y* or similar [1 mark]

- (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]**

Solution: *Comprehension, Knowledge*

In eager evaluation, the arguments to functions are always fully evaluated before the function is applied [2 marks]. In contrast, in lazy evaluation, the function is applied first, before its arguments are evaluated [2 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]

Solution: *Comprehension, Application*

Neither normal form, nor WHNF.

ii. `1 : 2 : []`

[2 Marks]

Solution: *Comprehension, Application*

Normal form.

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

[4 Marks]

continued

Solution: *Comprehension, Application*
WHNF.

- (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]**

Solution: *Synthesis, Analysis*

With outermost evaluation, arguments to a function are only needed when they are demanded by another computation. In some circumstances, this can lead to Haskell building a very large expression graph before collapsing it to generate a final result [2 marks]. A prototypical example is `foldl` which generates an expression graph whose size grows linearly in the size of the list it is processing. Strict evaluation can help here by forcing Haskell to evaluate intermediate results immediately, thus controlling the size of the expression graph [2 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]**

Solution: *Synthesis, Application, Analysis*

A good answer here could touch on a number of points, the below is not exhaustive, and there is not one “right” answer. If you come up with “wrong” reasons, those are likely to get some marks too if the explanation seems reasonable.

A functional language, or interface, is a nice place to start when implementing program transformations, and automatic parallelism. The primary reason is that such languages offer *referential transparency*: it is always safe to substitute an expression by its definition.

This enables frameworks to apply transformations based on equational reasoning. For example, in a referentially transparent language, given a function call like

```
map f some_list
```

we can guarantee that transformation to the following is safe

```
concat [map f list_a, map f list_b]  
  where (list_a, list_b) = splitAt midpoint some_list
```

the two sublists can then be processed in parallel. Performing this reasoning doesn't need to know anything about `f`. In contrast, in a language like C, optimising the equivalent computation would need to inspect the body of `f` to ensure that there were no side-effects.

Relatedly, verification of correctness of our transformations is much simpler. We must prove that the equational law we state is valid once, and again by referential transparency it is valid all the time. In contrast, for a language that can mutate state in place, each potentially parallel piece of code must be analysed on its own to ensure that transformations are safe. This makes analysing the correctness much more complicated.

As a consequence, compilers often take the (possibly imperative, non-functional) input, and transform to a functional representation that is easier to reason about.

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]

Solution: *Comprehension, Knowledge, Analysis*

(a) A private class variable with a public property means that other classes must access the variable through the property. Therefore, one can implement customised process to monitor the access of the variable, such as data validation before setting the variable. A public class variable cannot enforce access controls. [4 marks]

(b) A class can only inherit from one abstract class but can implement multiple interfaces. [2 marks]

Abstract classes can have class variables, but interfaces cannot. [2 marks]

Abstract classes can have both abstract and non-abstract class methods, but interfaces can only have abstract methods. [3 marks]

Other answers allowed.

(c) Similarity: Both method hiding and method overriding aims at re-implementing a method of the parent class from a child class. [3 marks]

Difference: In method hiding, the child version and parent version of the method co-exist and a child object can call both versions of the method. In method overriding, the parent version of the method cannot be called by a child object anymore. [3 marks]

Other answers allowed.

(d) Windows components in WPF are generally developed under a hierarchical inheritance tree. For example, when creating a custom window that holds other user interface elements, the custom window inherits the window class, which comes with existing implementation of events, variables and methods. [4 marks]

Other answers allowed.

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

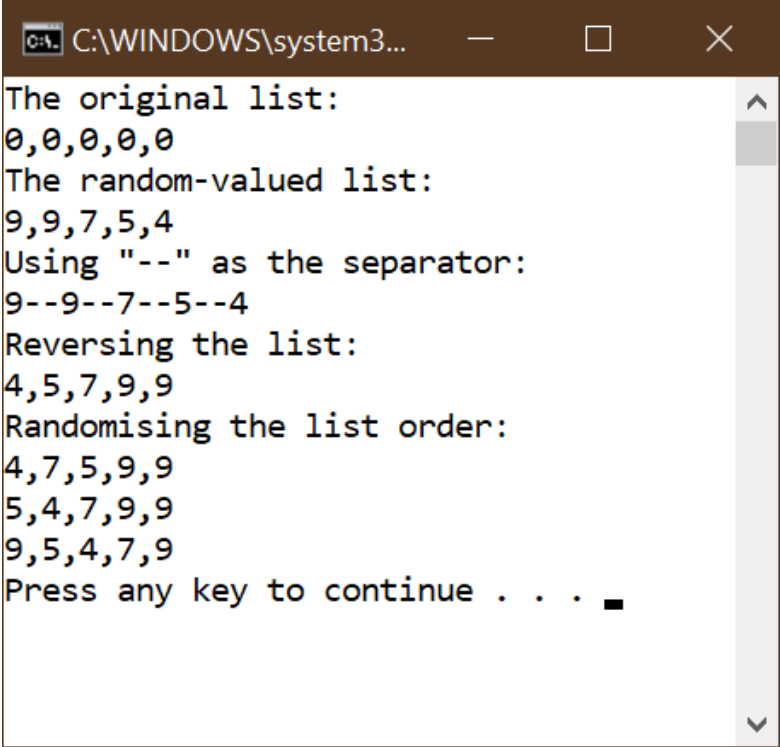
```
public class NumberList
{
    private int iSize;
    private List<int> lstI = new List<int>();
    private Random ran = new Random();

    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]**

Solution: *Comprehension, Analysis, Application*

(a) Marks for header and proper return (no return for void) [1 marks], the proper use of random function and proper logics [3 marks]. Other working solutions are also acceptable.

The function iterates through the list and assign each item with a random value between 1 and 9 inclusive.

Solution

```
public void RandomValues()
{
    for (int i=0; i<lstI.Count; i++)
    {
        lstI[i] = ran.Next(1, 10);
    }
}
```

(b) Marks for header and proper return (no return for void) [2 marks], proper logic generating the required output [4 marks]. Other working solutions are also acceptable.

The function iterates through list. For each iteration, it prints the value of the list item. If the item is not the last one, it prints the separator. After the loop, it prints the line break.

Solution

```
public void Print(string strSperator)
{
    for (int i = 0; i < lstI.Count; i++)
    {
        Console.Write(lstI[i]);
        if (i != lstI.Count - 1)
            Console.Write(strSperator);
    }
    Console.WriteLine();
}
```

(c) Marks for header and proper return (no return for void) [2 marks], proper logic [6 marks]. Other working solutions are also acceptable.

The function creates a new list. It iterates through lstI from the last item to the first, and insert the items one by one to the new list. It finally sets lstI to the new list.

Solution

```
public void Reverse()
{
    List<int> lstIReverse = new List<int>();
    for (int i = lstI.Count - 1; i >= 0; i--)
        lstIReverse.Add(lstI[i]);
    lstI = lstIReverse;
}
```

(d) Marks for header and proper return (no return for void) [2 marks], the proper use of random function [3 marks], proper logic [7 marks]. Other working solutions are also acceptable.

The function creates a new list. Inside a loop that continues until lstI is empty, it creates a random number between 0 and the size of lstI as an index, adds the item of lstI with the index to the new list, and removes the item from lstI. It finally sets lstI to the new list.

Solution

```
public void RandomOrder()
{
    int iRandom;
    List<int> lstIRandom = new List<int>();

    while (lstI.Count > 0)
    {
        iRandom = ran.Next(0, lstI.Count);
        lstIRandom.Add(lstI[iRandom]);
        lstI.RemoveAt(iRandom);
    }
    lstI = lstIRandom;
}
```