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 $[x_0, x_1, x_2, \ldots, x_{n-1}]$, `scan` should return the list $[y_0, y_1, y_2, \ldots, y_{n-1}]$ where $y_0 = x_0$, $y_1 = x_0 + x_1$, and generally $y_j = \sum_{i=0}^{j} x_j$.

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]**

this question is continued on the next page

(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]**