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]**

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]
```

continued

```
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.

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]**

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.

continued

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.