CS2560

UNIVERSITY OF WARWICK

Second Year Examinations: Summer 2016

Functional Programming

Time allowed: 2 hours.

Answer **FIVE** questions in total. These must be **Question 1**, any **TWO** questions from Section A, and any **TWO** questions from Section B.

Read carefully the instructions on the answer book and make sure that the particulars required are entered on each answer booklet.

Calculators are not allowed.

The following Haskell functions are set as exercises in this exam either by providing definitions for you to use or having you prepare your own: `foldl`, `last`, `length`, `product`, `reverse`, and `sum` . Thus the implementations of these functions as available in the Standard Prelude must **NOT** be used in this exam.

---

1. Answer each of the two parts of this question in about 100 words.

   (a) Functional programming uses a *declarative* paradigm. Explain what this is. Compare and contrast the imperative programming paradigm with a declarative one. [10]

   (b) Functional programming uses *higher order functions* (HOFs). Explain what these are. What are the strengths and weaknesses of HOFs? [10]

Section A Answer **TWO** questions

The theme of Section A is functional programming using **first order function definitions**.

2. Give a first order definition for each of the following Haskell functions.

   (a) `nonspace :: [Char] -> Int` which returns the number of non space characters in any given finite list of integers. For example, `nonspace "fun"` evaluates to `3` , and `nonspace "fun prog"` evaluates to `7`. [6]

   (b) `isort :: Ord a => [a] -> [a]` which sorts a given list (over an ordered type) into ascending order using the well known *insertion sort* algorithm. This algorithm is tail recursive, finding the rightful place for the head of a non-empty list in the recursively sorted tail. For example, `isort [4,3,1]` evaluates to `[1,3,4]`. [8]

   (c) `sum :: [Int] -> Int` which sums all of the integers in a given finite list using an **iterative** algorithm. For example, `sum [6,3,7]` evaluates to `16`. [6]

   Note that each of the three functions in Question 2 can be defined without reference to the other two.

3. This question is about how to model *polymorphic finite binary trees* in Haskell using first order function definitions. Using the `data` declaration,

```
data Tree a = Leaf a | Node (Tree a) a (Tree a)
```

   we can for example define,

```
test :: Tree Char
test = Node (Leaf 'a') 'b' (Node (Leaf 'd') 'c' (Leaf 'e'))
```

   Define each of the following functions.

   (a) A *show* function `showt :: Tree a -> [Char]` which represents an enumeration of the nodes in string format using a *left to right depth first* traversal of the nodes of a given tree. For example, `showt test` evaluates to `"abdce"` . [6]

   (b) `depth :: Tree a -> Int` which returns the *depth* of a given tree. This being defined to be the length of a longest branch, which is defined to be the number of nodes in such a branch. For example, `depth test` evaluates to `3`. [6]

   (c) `equalt :: (Tree a, Tree a) -> Bool` which returns `True` if & only if two given trees are *equal.* That is, if their nodes are equal, and if recursively their subtrees are equal. [8]

4. Define each of the following functions in Haskell using a first order function definition and an **iterative** approach.

   (a) `match :: [Char] -> Bool` is a function to parse simple expressions of the following form. Each expression may contain the characters `'('`, `')'`, and a lower case letter in the range `'a'` to `'z'`. For an expression `e` to be correct, that is, for `match e` to evaluate to `True` , the brackets must *match up* in the usual way. For example, `match ""` , `match "(x)"` , and `match "(x(y))"` each evaluates to `True` . And, `match ")("` , `match "(()"` , and `match "(+)"` each evaluates to `False` . [10]

   (b) `final :: [a] -> a` is a function which for the empty list `[]` returns a suitable error message, and for a non-empty list returns the final item. For example, `final "CS256"` evaluates to `'6'` , and `final [1,3,6]` evaluates to `6`. [10]

Section B Answer **TWO** questions

The theme of Section B is functional programming using **higher order function definitions**.

5. (a) A *for loop* is an iterative construct which from a given initial state repeatedly applies a given action for each index in a range of integers. In Haskell we can declare the name and type of a *for loop* as follows.

```
for :: Int -> Int -> (a -> a) -> (a -> a)
```

   That is, `for n m f s` repeatedly applies a function `f` starting from an initial state `s` and an implicit *counter* initially valued `n` , incrementing the counter by `1` until it exceeds `m`. Give a definition in Haskell for the *for loop* `for` . [6]

   (b) *For loops* are found in many programming languages, being very efficient for simple iteration. Using your Haskell `for` loop define the function `mult :: Int -> Int` which multiples $1 \times 2 \times \ \ldots \ \times n$ for any given integer $n \geq 1$. For example, `mult 4` evaluates to `24` . [6]

   (c) A key role for higher order functions in functional programming is to design language constructs which may not be provided by your favourite language. For example, there are many variations on a *for loop* in different languages. Design a generalisation of `for` called `forge` which adds another Curried argument to specify what function is to be used to increment the counter. For example, `forge` with increment function `(+ 1)` is in effect `for` . [8]

6. This question is about defining and using *folding from the left* in functional programming.

   (a) Define the Haskell function,

   ```
   foldl :: (a -> b -> a) -> a -> [b] -> a
   ```

   for *folding from the left*. [5]

   (b) Using `foldl` define the Haskell function `length :: [a] -> Int` which returns the length of a list. For example, `length "abc"` evaluates to `3` . [5]

   (c) Using `foldl` define the Haskell function `reverse :: [a] -> [a]` which reverses the order of the items in a finite list. For example, `reverse "abc"` evaluates to `"cba"` . [5]

   (d) Using `foldl` define the Haskell function `map :: (a -> b) ->[a] -> [b]` such that `map f l` applies `f` to each item in `l` . For example, `map (* 2) [1,3,5]` evaluates to `[2,6,10]` . [5]

7. The theme of Question 7 is the untyped λ-calculus. That is, the calculus of anonymous functions such as $\lambda n\ .\ (+1)\ n$ which form the logical foundation of functional programming.

   (a) Give a hand coded evaluation for the lambda expression, $(+)\ ((\lambda n\ .\ (+)\ n\ 2)\ 3)\ ((\lambda n\ .\ (*)\ 3\ ((-)\ 6\ n))\ 4)$. [12]

   (b) The so-called Y *combinator* is defined in the original notation of untyped $\lambda-$calculus by, $Y\ =\ \lambda f\ .\ (\lambda x\ .\ f(x(x)))\ (\lambda x\ .\ f(x(x)))$. For any function g prove that $Y(g)\ \rightsquigarrow\ g(Y(g))$. That is, $Y(g)$ reduces to $g(Y(g))$. [8]