Final Exam
Informatics 1 – Introduction to Computation
Functional Programming Tutorial 10

Wadler

Week 11
due 4pm Wednesday 2 December 2020
tutorials on Friday 4 December 2020

Attendance at tutorials is obligatory; please send email to lambrose@ed.ac.uk if you cannot join your assigned tutorial.

Good Scholarly Practice: Please remember the good scholarly practice requirements of the University regarding work for credit. You can find guidance at the School page

http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct.

This also has links to the relevant University pages. Please do not publish solutions to these exercises on the internet or elsewhere, to avoid others copying your solutions.

UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

INFORMATICS 1 — INTRODUCTION TO COMPUTATION

Wednesday 25þ November 2020

16:00 to –:–

INSTRUCTIONS TO CANDIDATES

1. **Note that ALL QUESTIONS ARE COMPULSORY.**
2. **DIFFERENT QUESTIONS MAY HAVE DIFFERENT NUMBERS OF TOTAL MARKS. Take note of this in allocating time to questions.**
3. **CALCULATORS MAY NOT BE USED IN THIS EXAMINATION**

```
div, mod :: Integral a => a -> a -> a
even, odd :: Integral a => a -> Bool
(+), (*), (-), (/) :: Num a => a -> a -> a
(<), (<=), (>), (>=) :: Ord => a -> a -> Bool
(==), (/=) :: Eq a => a -> a -> Bool
(&&), (||) :: Bool -> Bool -> Bool
not :: Bool -> Bool
max, min :: Ord a => a -> a -> a
isAlpha, isAlphaNum, isLower, isUpper, isDigit :: Char -> Bool
toLower, toUpper :: Char -> Char
ord :: Char -> Int
chr :: Int -> Char
```

Figure 1: Basic functions

```
sum, product :: (Num a) => [a] -> a
sum [1.0,2.0,3.0] = 6.0
product [1,2,3,4] = 24

maximum, minimum :: (Ord a) => [a] -> a
maximum [3,1,4,2]  =  4
minimum [3,1,4,2]  =  1

concat :: [[a]] -> [a]
concat ["go","od","bye"]  =  "goodbye"

(!!) :: [a] -> Int -> a
[9,7,5] !! 1  =  7

head :: [a] -> a
head "goodbye" = 'g'

init :: [a] -> [a]
init "goodbye" = "goodby"

takeWhile :: (a->Bool) -> [a] -> [a]
takeWhile isLower "goodBye" = "good"

dropWhile :: (a->Bool) -> [a] -> [a]
dropWhile isLower "goodBye" = "Bye"

elem :: (Eq a) => a -> [a] -> Bool
elem 'd' "goodbye" = True

zip :: [a] -> [b] -> [(a,b)]
zip [1,2,3,4] [1,4,9] = [(1,1),(2,4),(3,9)]
```

```
and, or :: [Bool] -> Bool
and [True,False,True] = False
or [True,False,True] = True

reverse :: [a] -> [a]
reverse "goodbye" = "eybdoog"

(++) :: [a] -> [a] -> [a]
"good" ++ "bye" = "goodbye"

length :: [a] -> Int
length [9,7,5]  =  3

tail :: [a] -> [a]
tail "goodbye" = "oodbye"

last :: [a] -> a
last "goodbye" = 'e'

take :: Int -> [a] -> [a]
take 4 "goodbye" = "good"

drop :: Int -> [a] -> [a]
drop 4 "goodbye" = "bye"

replicate :: Int -> a -> [a]
replicate 5 '*' = "*****"
```

Figure 2: Library functions

For this exam, *basic functions* refers to functions in Figure 1 and *library functions* refers to functions in Figure 2 on the preceding page.

1. (a) Write a function `f :: [String] -> String` that finds the smallest string consisting only of lower-case letters and with length less than six, where strings are taken in dictionary order. If every string has at least six characters or contains a character that is not a lower-case letter, return `"zzzzz"`. For example:

```
f ["a","bb","ccc","dddd","eeeee","ffffff"] = "a"
f ["uuuuuu","vvvvv","wwww","xxx","yy","z"] = "vvvvv"
f ["Short","longer","???"] = "zzzzz"
```

Use *basic functions*, *list comprehension*, and *library functions*, but *not recursion*. Credit may be given for indicating how you have tested your function. [*12 marks*]

(b) Write a function `g :: [String] -> String` that behaves like `f`, this time using *basic functions* and *recursion*, but *not list comprehension* or *library functions* from Figure 2 (other than length). Credit may be given for indicating how you have tested your function. [*12 marks*]

(c) Write a function `h :: [String] -> String` that also behaves like `f`, this time using one or more of the following higher-order functions:

```
map     :: (a -> b) -> [a] -> [b]
filter  :: (a -> Bool) -> [a] -> [a]
foldr   :: (a -> b -> b) -> b -> [a] -> b
```

You may use *basic functions* but do *not* use *recursion*, *list comprehension* or *library functions* from Figure 2 (other than length). Credit may be given for indicating how you have tested your function. [*12 marks*]

2. (a) Write a function `i :: [a] -> [a] -> [a]` that takes two non-empty lists and returns the tail of the first followed by the head of the second. For example:

```
i "abc" "def" = "bcd"
i "def" "ghi" = "efg"
i "ghi" "abc" = "hia"
```

You may use any functions you wish. [*4 marks*]

(b) Write a function `j :: [[a]] -> [[a]]` that takes a non-empty list of non-empty lists, and moves the first element of each list to become the last element of the preceding list. The first element of the first list becomes the last element of the last list. For example:

```
j ["abc","def","ghi"] = ["bcd","efg","hia"]
j ["once","upon","a","time"] = ["nceu","pona","t","imeo"]
j ["a","b","c"] = ["b","c","a"]
j ["a"] = ["a"]
```

Use *basic functions*, *list comprehension*, and *library functions*, but *not recursion*. You may use your answer to 2(a). Hint: you may wish to use `i` twice in your solution. Credit may be given for indicating how you have tested your function. [*14 marks*]

(c) Write a function `k :: [[a]] -> [[a]]` that behaves like `j`, this time using *basic functions* and *recursion*, but *not list comprehension* or *library functions*. You may use your answer to 2(a). Credit may be given for indicating how you have tested your function. [*14 marks*]

3. The following data type represents propositions with two possible variables, `X` and `Y`, constants true and false, and connectives for negation, conjunction, disjunction, and implication.

```
data Wff = X
         | Y
         | Tr
         | Fa
         | Not Wff
         | Wff :&: Wff
         | Wff :|: Wff
         | Wff :->: Wff
```

The template file provides instances

```
(==) :: Wff -> Wff -> Bool
show :: Wff -> String
```

to compare two propositions for equality and to convert a proposition into a readable format. It also provides code that enables QuickCheck to generate arbitrary values of type `Wff`, to aid testing.

(a) Write a function `eval :: Bool -> Bool -> Wff -> Bool` that takes boolean values for `X` and `Y` and returns the boolean value of the proposition. For example:

```
eval False False ((X :->: Y) :&: (Not Y :|: X))  =  True
eval False True  ((X :->: Y) :&: (Not Y :|: X))  =  False
eval True  False ((X :->: Y) :&: (Not Y :|: X))  =  False
eval True  True  ((X :->: Y) :&: (Not Y :|: X))  =  True
```

Credit may be given for indicating how you have tested your function. [*8 marks*]

(b) We call a proposition *simple* if it does not contain true or false as proper subterms. Write a function `simple :: Wff -> Bool` that determines whether a proposition is simple. For example:

```
simple Tr                             =  True
simple Fa                             =  True
simple ((Tr :|: X) :->: (Tr :&: Y))   =  False
simple ((X :|: Fa) :->: (Y :&: Fa))   =  False
simple ((X :&: Y) :->: (X :|: Y))     =  True
```

Credit may be given for indicating how you have tested your function. [*8 marks*]

QUESTION CONTINUES ON NEXT PAGE

(c) Write a function `simplify :: Wff -> Wff` that converts a proposition to an equivalent proposition which is simple, using the following laws:

```
Not Tr      =  Fa
Not Fa      =  Tr
Fa :&: p    =  p :&: Fa    =  Fa
Tr :&: p    =  p :&: Tr    =  p
Fa :|: p    =  p :|: Fa    =  p
Tr :|: p    =  p :|: Tr    =  Tr
Fa :->: p   =  p :->: Tr   =  Tr
Tr :->: p   =  p
p :->: Fa   =  Not p
```

For example:

```
simplify Tr                              =  Tr
simplify Fa                              =  Fa
simplify ((Tr :|: X) :->: (Tr :&: Y))    =  Y
simplify ((X :|: Fa) :->: (X :&: Fa))    =  Not X
simplify ((X :|: Y) :->: (X :&: Y))      =  ((X :|: Y) :->: (X :&: Y))
```

Credit may be given for indicating how you have tested your function. [*16 marks*]