Informatics 1 – Introduction to Computation
Functional Programming Tutorial 10

Sannella, Wadler

Week 11
due 12:00 Tuesday 28 November 2023
tutorials on Thursday 30 November and Friday 1 December 2023

You will not receive credit for your coursework unless you attend the corresponding tutorial session. Please email your tutor 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 Please do not publish solutions to these exercises on the internet or elsewhere, to avoid others copying your solutions.

```
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

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

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

all, any :: (a -> Bool) -> [a] -> Bool
all odd [1,2,3] = False
any odd [1,2,3] = True

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 2: Library functions

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

1. (a) Say that a string is *ok* if it consists only of lower case letters and has length less than six. Write a function `ok :: String -> Bool` that identifies ok strings. For example:

```
ok "a"      = True
ok "zzzzz"  = True
ok "Short"  = False
ok "longer" = False
ok "???"    = False
```

You may use any functions you wish.

(b) Write a function `f :: [String] -> String` that finds the smallest ok string, where "smallest" refers to dictionary order. If no string is ok, 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*. You may use your answer to 1(a).

(c) 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. You may use your answer to 1(a).

(d) 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. You may use your answer to 1(a).

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.

(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.

(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* from Figure 2. You may use your answer to 2(a).

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 Prop = X
          | Y
          | T
          | F
          | Not Prop
          | Prop :&&: Prop
          | Prop :||: Prop
          | Prop :->: Prop
```

The template file provides instances

```
(==) :: Prop -> Prop -> Bool
show :: Prop -> 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 `Prop`, to aid testing.

(a) Write a function `eval :: Bool -> Bool -> Prop -> 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
```

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

```
simple T                              =  True
simple F                              =  True
simple ((T :||: X) :->: (T :&&: Y))  =  False
simple ((X :||: F) :->: (Y :&&: F))  =  False
simple ((X :&&: Y) :->: (X :||: Y))  =  True
```

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

```
Not T      =  F
Not F      =  T
F :&&: p   =  p :&&: F  =  F
T :&&: p   =  p :&&: T  =  p
F :||: p   =  p :||: F  =  p
T :||: p   =  p :||: T  =  T
F :->: p   =  p :->: T  =  T
T :->: p   =  p
p :->: F   =  Not p
```

For example:

```
simplify T                                =  T
simplify F                                =  F
simplify ((T :||: X) :->: (T :&&: Y))  =  Y
simplify ((X :||: F) :->: (X :&&: F))  =  Not X
simplify ((X :||: Y) :->: (X :&&: Y))  =  (X :||: Y) :->: (X :&&: Y)
```

Optional: Parsing Propositions

> Following the Common Marking Scheme, a student with good mastery of the material is expected to get 3/4 points. This section is for demonstrating exceptional mastery of the material. It is optional and worth 1/4 points.

The template file for solutions for this part of the coursework is `Prop.hs`. The file defines a datatype for propositions, the same as the definition above, and an appropriate `show` function. It makes reference to the file `MyParser.hs` which contains code for monadic parsing.

Exercise 1

Write a parser

```
parseProp :: Parser Prop
```

that can read a proposition from a string. It should be the inverse to the provided show function.

Exercise 2

Write a QuickCheck property

```
prop_roundtrip :: Prop -> Bool
```

that checks that if you show a proposition to convert it to a string and parse the string to convert it back to a proposition then the result is the original proposition.