G6021: Comparative Programming

Exam Practice 2

1. (a) Give an example for each of the following:

   i. a λ-term that is in normal form but does not have a type.

   ii. a λ-term that has a normal form but does not have a type.

   iii. a λ-term that has a normal form and does have a type.

   Answer: Let $I = \lambda x.x, \Delta = \lambda x.xx$.

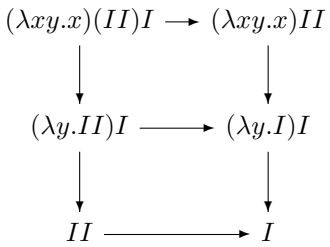
   i. Δ

   ii. $I\Delta$

   iii. II

   (b) Give the β-reduction graph of the λ-term $(\lambda xy.x)(II)I$, where $I = \lambda x.x$.

   Answer: A graph showing all possible reductions of a λ-term.



2. In the Rock-Paper-Scissors game, the rules are that *paper* beats *rock*, *rock* beats *scissors*, and *scissors* beats *paper*.

   (a) Define a type `RPS` to represent rock, paper and scissors as used in the game in both Haskell and Java. In no more that 10 sentences, compare the two.

   Answer:

```
data RPS = Rock  | Paper | Scissors deriving (Show)

class RPS {}
class Rock extends RPS {}
class Paper extends RPS {}
class Scissors extends RPS {}
```

   Haskell: disjoint union types are directly implementable as shown. Types are immediately available (both input and output).
   Java: need to use the object structure. Each class in separate file makes maintenance of methods more difficult. More verbose than Haskell. Quite a lot of effort needed to build objects of this new type to test code, etc.

   (b) Write a function in Haskell: `beats :: RPS -> RPS -> Bool` that encodes that paper beats rock, rock beats scissors, and scissors beats paper.

   Answer:

```
beats :: RPS -> RPS -> Bool
beats Paper Rock = True
beats Rock Scissors = True
beats Scissors Paper = True
beats _ _ = False
```

(c) Suppose you wanted to write a function/method to return two values (say a pair of integers). Outline how you would do this in both Haskell and Java.

Answer: Haskell has pairs (and triples, etc.) built-in, so there is nothing new needed for Haskell. For Java, we need to create a new type, for example at least this is needed:

```
class Pair {
  int fst, snd;
}
```

3. (a) Write two recursive functions in Haskell `sumAll` that returns the sum of all the elements of a list of numbers and `prodAll` that returns the product of all the elements in a list of numbers.

Answer:

```
sumAll [] = 0
sumAll (h:t) = h+sumAll t

prodAll [] = 1
prodAll (h:t) = h*prodAll t
```

(b) Write in Haskell the higher-order function `foldr` which takes three parameters: a function, a value and a list. Include the type of the function `foldr` in your answer, and give two examples of use of `foldr` to compute the sum and product of a list of numbers (as in the previous question).

Answer:

```
foldr :: (t -> t2 -> t2) -> t2 -> [t] -> t2
foldr f b [] = b
foldr f b (h:t) = f h (foldr f b t)

foldr (*) 1 [1,2,3,4]
foldr (+) 0 [1,2,3,4]
```

(c) Using the following function as an example

```
g (x,y) = x+y
```

Explain the concept of currying a function. Curry the function `g` and give the type of the resulting function.

Answer:

```
f :: Int -> Int -> Int
g x y = x+y
```