G6021: Comparative Programming

Exercise Sheet 2

1 Function arguments

If a function needs two arguments, it can take them one at a time or both together, as illustrated by the following functions:

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

We say that `f` is a *curried* version of `g`.

1. What are the types of `f` and `g`? Test the above two function definitions, including the type definition of each function.

   Answer:

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

   g :: (Int,Int) -> Int
   g (x,y) = x+y
   ```

   Note that if we do not give the types as above, and ask Haskell to derive them, you will get an answer something like:

   ```
   f :: Num a => a -> a -> a
   g :: Num a => (a, a) -> a
   ```

   These types are given in terms of Haskell class types. You can read these types in the following way, using `f` as the example: if `a` is a member of the type class `Num`, then the type is `a -> a -> a`. This means that the function has a number of different types of the shape `a -> a -> a`. One example type that is in `Num` is `Int`, try to find out the others!

2. Consider a function that returns the sum of three integer arguments. Try to write four alternative versions that take arguments in different ways (think about currying, and mixing styles). What are the types of your functions?

   Answer:

   ```
   sum1 :: Int -> Int -> Int -> Int
   sum1 x y z = x+y+z

   sum2 :: Int -> (Int, Int) -> Int
   sum2 x (y,z) = x+y+z

   sum3 :: (Int, Int) -> Int -> Int
   sum3 (x,y) z = x+y+z

   sum4 :: (Int, Int, Int) -> Int
   sum4 (x,y,z) = x+y+z
   ```

3. Write a function that takes four arguments. The body of the function can be anything, but must use the 1st argument once, the 2nd argument twice, the 3rd argument once and ignores the 4th argument. What is the type of this function?

   Answer:

   ```
   f :: Int -> Int -> Int -> t -> Int
   f a b c d = a+b+b+c
   ```

   What do you think `t` means in the above type

4. Two built-in functions provided by the Haskell system are:

   ```
   curry :: ((a, b) -> c) -> a -> b -> c
   uncurry :: (a -> b -> c) -> (a, b) -> c
   ```

   By looking carefully at the types of the functions `f` and `g` above, show how you can convert `f` to take one argument (a pair of numbers) and convert `g` to take 2 arguments (one at a time).

   Answer: Example: to add 3 and 4:

   ```
   curry g 3 4
   uncurry f (3,4)
   ```

5. Write your own version of `curry` and `uncurry`.

   Answer:

   ```
   mycurry f x y = f (x,y)
   myuncurry f (x,y) = f x y
   ```

2 Lists in Haskell

1. Enter `[1,2,3,4,5]`, and find out the type of this expression.

   Answer:

   ```
   Prelude> [1,2,3,4,5]
   [1,2,3,4,5]
   Prelude> :t it
   it :: [Integer]
   ```

2. Find two other ways to write (or generate) the list `[1,2,3,4,5]`.

   Answer:

   ```
   Prelude> 1:2:3:4:5:[]
   [1,2,3,4,5]
   Prelude> [1..5]
   [1,2,3,4,5]
   ```

3. What is generated by `[x | x<-[1..], x<6]`? Think about it before testing with the interpreter.

   Answer: `[1..]` will generate an infinite list of integers starting from 1, and the list comprehension will select all the integers less than 6. However, the program will not terminate as all the integers must be tested. The output is therefore:

```
Prelude> [x | x<-[1..], x<6]
[1,2,3,4,5
```

and control-c is needed to stop the computation.

4. Write a function `inorder` that will test if a list of integers is sorted in ascending order. Hint: The preferred solution would use pattern matching. Some examples of patterns are:

   - `[]` : the empty list.
   - `[x]` : a list with exactly 1 element (alternative notation: `x:[]`).
   - `(x:t)` : a list with at least one element.
   - `(x:y:t)` : a list with at least two elements, etc.

   Answer:

   ```
   inorder [] = True
   inorder [x] = True
   inorder (x:y:t) = x<=y && inorder (y:t)
   ```

5. Write a function `insert` that takes an integer `x` and a sorted list, and returns the sorted list with `x` inserted in the correct place.

   Example: `insert 3 [1,2,4,5] = [1,2,3,4,5]`

   Answer:

   ```
   insert x [] = [x]
   insert x (h:t)
     | x<=h = x:h:t
     | otherwise = h:insert x t
   ```

   We use `otherwise` to catch all when the other cases fail. How do you think `otherwise` is implemented in Haskell? See footnote for answer[1].

6. Using the previous function or otherwise, write a sort function.

   Example: `sort [4,3,2,1] = [1,2,3,4]`

   Answer:

   ```
   sort [] = []
   sort (h:t) = insert h (sort t)
   ```

3 If you have time...

1. Write a function that returns the first n elements of a list (you can assume that the list is longer than n). Examples:

   ```
   first 3 [9,2,4,7,6,5] = [9,2,4]
   first 0 [1,2,3,4,5,6,7] = []
   first 10 [1..] = [1,2,3,4,5,6,7,8,9,10]
   ```

   (Hint: use the first parameter to countdown.)

   Answer:

   ```
   first n (h:t) = if n==0 then [] else h:first (n-1) t
   ```

[1] `otherwise` is a Boolean constant: `True`. It's possible to replace `otherwise` by `True` is any program, and `otherwise = True` is how it is defined in the standard library.

2. Write a function `filt` that takes a number n and a list of numbers, and returns the list with all the multiples of n removed. You can make use of a built-in function `mod` that gives the remainder after division (`mod 2 2 = 0`, `mod 3 2 = 1`).[2]

   Examples:

   ```
   filt 2 [1,2,3,4,5,6] = [1,3,5]
   filt 1 [1,2,3,4] = []
   filt 3 [1..10] = [1,2,4,5,7,8,10]
   ```

   Answer:

   ```
   filt n [] = []
   filt n (h:t) = if h ‘mod‘ n == 0 then filt n t else h:filt n t
   ```

3. Starting with the list `[2..]`, we can generate a list of prime numbers by repeatedly doing the following two steps:

   (a) Keep the first element of the list (say h) as this is a prime number.

   (b) Delete all multiples of h from the remaining list.

   Write a function `primes` that will generate all the prime numbers. Using your function `first` test your algorithm to produce the first 10, 100, 1000, etc. primes.

   Answer:

   ```
   primes (h:t) = h:primes(filt h t)
   ```

   ```
   first 10 (primes [2..])
   first 100 (primes [2..])
   first 1000 (primes [2..])
   ```

[2] 2 ‘mod‘ 2 is a way to use this function infix rather than prefix.