CSCE 330 Fall 2013
FINAL EXAM
Friday 2013-12-09 (Closed Book and Notes)

1 FP–25 points

1. (6 points) Write a function that multiplies by four the value of its argument plus one. Call it `functionone`. So, for example, `functionone:3` is `16.0`. (The "`.0`" appears if you use Carter Bays's FP interpreter.)

   Answer: `{functionone * @ [+ @ [id, %1], %4]}`

2. (4 points) Write a function that applies `functionone` to all elements of a sequence and give an example of its application to a sequence of three numbers. Do not give a name to the function.

   Answer: `& functionone`

   Answer: `& functionone: <1 2 3>` (or: `& functionone: <1 2 3>  => < 9.0 12.0 15.0 >`

3. (10 points) Write a function that computes the minimum of two numbers. Call is min.

   Answer: `{min  (< @ [1,2] -> 1 ; 2)}`

4. (5 points) Use the function in the previous exercise to write a function that computes the minimum of a non-empty sequence of numbers. Do not name the function. Give an example of use.

   Answer: `!min`, e.g., `!min: <1 2 3 5 4> gives 1`

2 Haskell—70 points

1. (5 points) A recursive function has two parts, the *basis* and the *inductive step*.

   (a) The basis computes the result for sufficiently small arguments, without making any recursive call.

   (b) The inductive step calls the function recursively, with smaller arguments.

   The following recursive function (which is intended to compute the product of a list of integers) breaks one of these two rules. Which one? Correct the error.

   ```
   prod2 :: Num a => [a] -> a
   prod2 ns = if null ns then 1 else head ns * prod2 ns
   ```

   Answer: The second (because the recursive call does not have a smaller argument); replace the last ns with (tail ns)

2. (5 points) Define a function `member` that tests for list membership. Define this function recursively, using patterns and a conditional expression in the recursive case.

   Answer:

   ```
   --member 2 [1,2,3] => True
   member x [] = False
   member x (y:ys) = if x == y then True else member x ys
   ```

3. (10 points) Using the `member` function of the previous exercise, define a function `intersect`, which takes two lists and computes their intersection. Your function should work correctly on lists that represent sets (i.e., lists without duplicates). Define this function recursively, using patterns and a conditional expression in the recursive case.

   Answer:

   ```
   --intersect [1,2,3] [5,2,1] => [1,2] (or [2,1]; order does not matter
   --intersect [1,2,3,2] [5,2,1] => [1,2,2]; order does not
   --  matter; multiplicity in the answer does not matter
   --intersect [1,2,3,2] [5,3,3,2,1] => [1,2,3,2]; multiplicity
   -- in the answer does not matter
   --intersect works as intersect on sets if lists do not have
   -- duplicates
   intersect ::  [a] -> [a] -> [a]
   intersect [] ys = []
   intersect (x:xs) ys = if member x ys then x : (intersect xs ys)
                                      else (intersect xs ys)
   ```

4. (25 points total) Define a function `doubleAll` that doubles all the entries in its argument list, which is a list of `Int`, in four different ways:

   (a) (5 points) a non-recursive function using list comprehension. (Name this doubleAll1.)

   (b) (5 points) a recursive function with a conditional expression. (Name this doubleAll2.)

   (c) (5 points) guarded equations. (Name this doubleAll3.)

   (d) (5 points) pattern matching. (Name this doubleAll4.)

   (e) (5 points) a non-recursive function that translates the FP function `& (* @ [%2,id])`. (Name this doubleAll5.)

   For each case, write the type of the function. (You do not need to be most general.)

   Answer:

```
doubleAll1 ns = [2 * n | n <- ns]

doubleAll2 ns = if null ns then [] else 2*(head ns) : doubleAll2 (tail ns)

doubleAll3 ns | null ns = []
              | otherwise = 2*(head ns) : doubleAll3 (tail ns)

doubleAll4 [] = []
doubleAll4 (n:ns) = 2*n : doubleAll4 ns

doubleAll5 = map (2*)
--also map (*2)

--types given below: note doubleAll5
Main> :type doubleAll1
doubleAll1 :: Num a => [a] -> [a]
Main> :type doubleAll2
doubleAll2 :: Num a => [a] -> [a]
Main> :type doubleAll3
doubleAll3 :: Num a => [a] -> [a]
Main> :type doubleAll4
doubleAll4 :: Num a => [a] -> [a]
Main> :type doubleAll5
doubleAll5 :: [Integer] -> [Integer]
Main>
```

5. (25 points) A library keeps track of books loaned to people in a database of pairs, (Person, Book). You have to write three functions:the first one looks up the books that a person has on loan; the second one updates the database when a person takes a book on loan; the third one updates the database when a person returns a book on loan. Here are the necessary declarations and definitions:

```
type Person = String
type Book   = String

type Database1 = [(Person, Book)]
```

Note that the type Database may conflict with a predefined type; this is why I used Database1 instead.

The code snippet below provides an example of a database, where Alice, Anna, and Robert are persons, while Asterix, Little Women, and Tintin are books:

```
exampleBase :: Database1
exampleBase = [("Alice","Tintin"), ("Anna","Little Women"),
               ("Alice","Asterix"), ("Robert","Tintin")]
```

(a) (5 points) Define a function `lookup` that takes a database and a person and returns the list of books that the person has on loan.

(b) (10 points) Define a function `makeLoan` that takes a database, a person, and a book, and returns a new database, with the person and the book pair added on.

(c) (10 points) Define a function `returnLoan` that that takes a database, a person, and a book, and returns a new database, with the (person, book) pair removed.

Answer:

```
--Database1 here because of possible conflicts with other database types
lookup :: Database1 -> Person -> [Book]
lookup dBase pers = [bk | (pers', bk) <- dBase, pers' == pers]

makeLoan :: Database1 -> Person -> Book -> Database1
makeLoan dBase pers bk = [(pers,bk)] ++ dBase

returnLoan :: Database1 -> Person -> Book -> Database1
returnLoan dBase pers bk =
     [pair | pair <- dBase, pair /= (pers,bk)]
```