G6021: Comparative Programming

Exercise Sheet 2

1 Functions

In Haskell, the following syntax is used for declaring functions:

- Function name
- List of parameters
- Output

For example,

```
square x = x*x
```

is a function that has a name `square`, a single parameter `x` and an output `x*x`.

Parameters are variables specified in a function definition and can be considered as placeholders. An argument, on the other hand, is a value passed when the function is called. Essentially, arguments take the place previously designated by the parameters. So, when we call a function:

```
square 6
```

Here, 6 is the argument, and the function is invoked. If a function has two parameters, it can take them one at a time or both together, as illustrated by the following alternatives:

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

2 Testing out the syntax

Guarded equations are a useful alternative syntax for conditionals. The following is a slight variant of an example given in the notes:

```
min2 x y
 | x < y = x
 | x > y = y
```

1. Test this function, and find the problem with it (you should find that it does not work as expected). Fix the code so that it gives the correct answer in all cases.
2. Because the order of the equations is respected, usually the last condition should catch all the remaining cases. There is a special syntax for this so we don't have to think about it: the keyword **otherwise**. Test this out in the example above.
3. What is the type of **otherwise**? What is its value?

3 If you have time: Lists in Haskell

1. Enter `[1,2,3,4,5]`, and find out the type of this expression.
2. Find two other ways to write (or generate) the list `[1,2,3,4,5]`.
3. What is generated by `[x | x<-[1..], x<6]`? Think about it before testing with the interpreter.
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.
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]`
6. Using the previous function or otherwise, write a sort function.

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

4 Optional (for fun): comparison of languages

Consider the following Java classes, showcasing 3 alternative implementations of `(false␣||␣true)␣&&␣true`.

```
public class MyBooleans {
  public static boolean iteand(boolean x,boolean y) {
    if (x) { return y; } else { return false; }
  }
  public static boolean iteor(boolean x,boolean y) {
    if (x) { return true; } else { return y; }
  }
  public static boolean cndand(boolean x,boolean y) {
    return x ? y : false;
```

```
  }
  public static boolean cndor(boolean x,boolean y) {
    return x ? true : y;
  }
  public static TV ddand(TV x,TV y) { return x.ddand1(y); }
  public static TV ddor(TV x,TV y) { return x.ddor1(y); }
  public static TV tvf = new TVF();
  public static TV tvt = new TVT();

  public static void main(String[] args) {
    System.out.print(iteand(iteor(false,true),true));
    System.out.print(cndand(cndor(false,true),true));
    ddand(ddor(tvf,tvt),tvt).ddprint();
    System.out.println((false || true) && true);
  }
}
public class TV {
  public void ddprint() { }
  public TV ddand1(TV y) { return null; }
  public TV ddor1(TV y) { return null; }
}
public class TVF extends TV {
  public void ddprint() { System.out.print("false"); }
  public TV ddand1(TV y) { return new TVF(); }
  public TV ddor1(TV y) { return y; }
}
public class TVT extends TV {
  public void ddprint() { System.out.print("true"); }
  public TV ddand1(TV y) { return y; }
  public TV ddor1(TV y) { return new TVT(); }
}
```

Observe that each of the 3 alternative implementations (exemplified in the `main` by each yielding `true` as output), adheres to a different paradigm.

1. Each of the 3 implementations of `and` and `or` exploits a *branching-construct* from a different paradigm, each offered by Java. Identify the respective Java constructs employed (in the respective methods for `and` and `or`) and how they serve to implement the two operations.
2. Is it good or bad that Java supports different paradigms?
3. Extend the program to also implement negation `not` (in each of the 3 ways).