CPSC 429/529: Functional Programming
Carsten Schürmann
Date: January 17, 2001

Homework 1

January 30, 2001.
9:30am.

Guidelines

While we acknowledge that beauty is in the eye of the beholder, you should nonetheless strive for elegance in your code. Not every program which runs deserves full credit. Make sure to state invariants in comments which are sometimes implicit in the informal presentation of an exercise. If auxiliary functions are required, describe concisely what they implement. Do not reinvent wheels, and try to make your functions small and easy to understand. Use tasteful layout and avoid long winded and contorted code. None of the problems requires more than a few lines of Haskell or SML code.

Problem 1: Binomial numbers

Binomial numbers play an important role in combinatorics because they answer the question of how many subsets can be formed by picking r elements out of a set of n. There are several ways to calculate binomial numbers. One way is by using factorials, the other one by reading them out of Pascal's triangle. One of the goals of this problem is to show the equivalence between both definitions.

1. (5 points) The factorial function is defined as

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{otherwise} \end{cases}$$

   Write a Haskell program

```
fac :: int -> int.
```

2. (5 points) Binomial numbers are defined as

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

   Write a Haskell program

```
bn :: (int, int) -> int.
```

3. (10 points) A way to calculate binomial numbers which does not require factorials is to use Pascal's triangle. The triangle has a 1 at its top, every other number in the triangle is defined as the sum of the numbers to the right and to the left above. All numbers outside the triangle are 0.

$$\begin{array}{ccccccccccc} & & & & 1 & & & & & & n=0 \\ & & & 1 & & 1 & & & & & n=1 \\ & & 1 & & 2 & & 1 & & & & n=2 \\ & 1 & & 3 & & 3 & & 1 & & & n=3 \\ 1 & & 4 & & 6 & & 4 & & 1 & & n=4 \\ \dots & & \dots & & \dots & & \dots & & & & \end{array}$$

   Write a Haskell program to compute the Pascal number at position (n, k) where n denotes the line and k the position within the line (starting at 0).

```
pas :: (int, int) -> int
```

4. (20 points) Show by induction that `pas` is a correct implementation of the binomial numbers as defined using factorials above. Carefully state and check the boundary conditions! You don't need to type your proof in (writing it by hand and handing it in in class is usually faster and easier).

5. (15 points) Consider the sum $\binom{n}{0} + \ldots + \binom{n}{n}$ for an arbitrary n. This sum can be written in closed form. To find the closed form, write a Haskell function

```
sumAll :: int -> int
```

   Experiment with it and guess the closed form. Then prove by induction that your claim is correct.

Problem 2: Flattening lists

We consider and compare two implementations of the flattening operation on trees which may be of arbitrary but finite branching degree. A tree has always one root, but may have several children. In Haskell such a tree can be represented as the following datatype as

```
data Tree = Leaf Double
          | Node [Tree]
```

We say that a function *flattens* a tree, if it removes the structure but not the content of the leaves which is then returned in form of a list. The order of elements in the returned list must correspond to the order of the leaves.

1. (10 points) Write a function that flattens a tree. This should be the straightforward implementation, and may use the append function `++`.

   ```
   flatten1 :: Tree -> [Double]
   ```

2. (10 points) The first function may be very inefficient, because lists are traversed several times. Write another, more efficient, function that flattens a tree. But this time, instead of just one argument, we pass two arguments to the new flatten function. The first argument is the Tree to be flattened, and the second an accumulated list of results. Write your function in such a way, that you can also do Part 3.

   ```
   flatten2 :: (Tree , [Double]) -> [Double]
   ```

3. (30 points) Show formally that for any tree T, `flatten1` applied to T and `flatten2` applied to (T, `[]`) compute the same results. (Hint: Use proof by induction!)