185.A05 Advanced Functional Programming SS 21

Monday, 15 March 2021

Assignment 1

on Chapter 1, Chapter 2, and Chapter 8.5

Topics: Streams, Stream Programming, Generate/Prune Pattern, Memoization, Abstract Data Types

Submission deadline: Monday, 22 March 2021, 12am

Regarding the deadline for the second submission: Please, refer to „Hinweise zu Organisation und Ablauf der Übung“ available at the homepage of the course.

Important:

1. Carefully read and follow the instructions given in the complementary files of assignment 0. If you have any questions regarding them, ask your questions in the TISS forum. Following the instructions is paramount to ensure a smooth processing of your submitted file with the test system.
2. Store all functions to be written for this assignment in a top-level file named

   `Assignment1.hs`

   of your group directory. Comment your program meaningfully; use auxiliary functions and constants, where reasonable. The very same file name shall be used for the second submission of assignment 1.
3. *Do not use self-defined modules!* If you want to re-use functions (written for other assignments), copy them to your submission file. Import declarations for self-defined modules will fail: Only `Assignment1.hs` will be copied for the (semi-automatical) evaluation procedure, no other ones.

Programming assignments:

The infinite sequence of *Catalan numbers* starting with

$$1, 1, 2, 5, 14, 42, 132, \ldots$$

is relevant for many combinatorial problems. E.g, the n-th Catalan is the number of meaningful ways of arranging a collection of n pairs of parentheses, or equivalently it is the number of ways to draw n ‘mountains’ using n up strokes and n down strokes starting with an up stroke from level 0 and ending with a down stroke again at level 0. For illustration, consider:

```
All well-formed sequences of n=0,1,2,3 pairs of parentheses:
n=0: { }
n=1: { () }
n=2: { ()(), (()) }
n=3: { ()()(), (()()), ((())), (())(), ()(()) }
```

```
The 5 mountains with 3 up and down strokes:

          /\
         /  \        /\          /\/\         /\
        /    \      /  \/\      /    \      /\/  \     /\/\/\
```

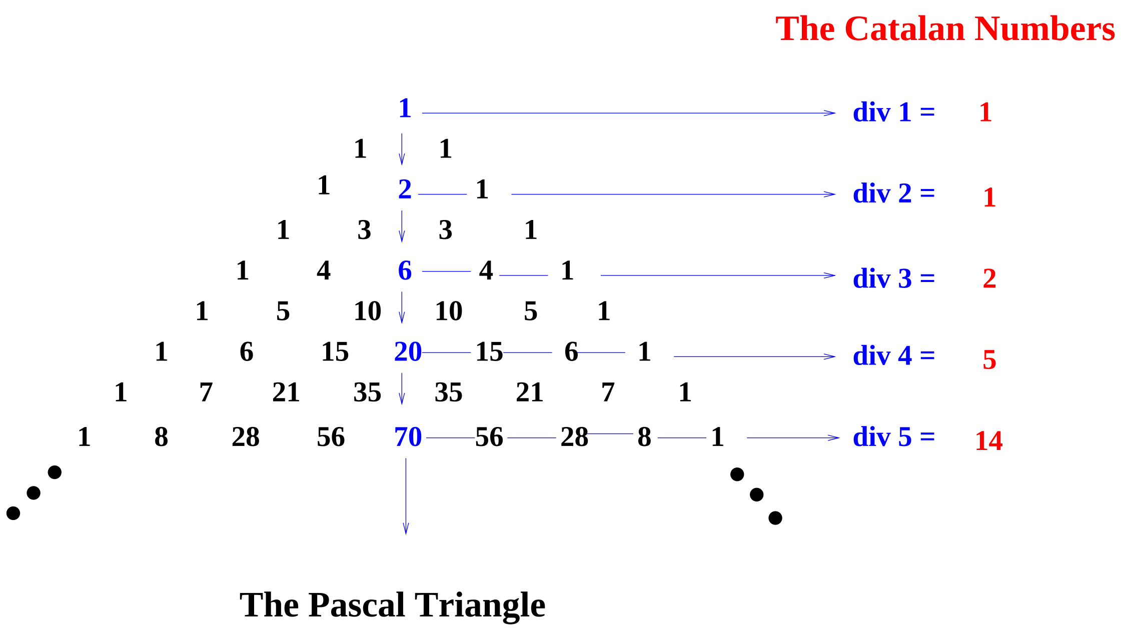
The below figure suggests computating the stream of Catalan numbers combining a *generator*, a *filter*, and a *transformer*:

- A generator g computing the Pacal triangle, also known as the arithmetic triangle.
- A filter f grabbing the middle numbers of every second row of the Pascal triangle (which are of odd length).
- A transformer t dividing the elements of the stream of numbers yielded by (the composition of g and) f elementwise by the elements of the stream of natural numbers (beginning with 1).

Numbering the rows in the Pacal triangle from top to(wards) bottom beginning with 1, the entry $C(n+1, k+1)$ in the Pascal triangle at row $n+1$ and column $k+1$ can be computed using the recurrence equation:

$$C(n+1, k+1) = C(n,k) + C(n, k+1) \quad \text{for } n = 1, 2, 3, \ldots; \; k = 1, \ldots n-1$$

Intuitively, this equation says that a number of the Pascal triangle is the sum of the two numbers sitting immediately on top of it (note, there is only one sitting on top of numbers sitting in the outermost diagonals of the Pascal triangle).



1. Implement a generator `g` computing the stream of (finite) rows of the Pascal triangle as a nullary co-recursive Haskell function:

   ```
   type Nat1 = Integer
   g :: [[Nat1]]
   ```

   Hence, evaluating `g` shall start with the lists:

   ```
            g ->> [ [1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1],...
   ```

2. Implement a filter `f` which applied to `g` yields the the stream of numbers sitting in the middle of every second row of stream `g` as a one-ary co-recursive Haskell function, i.e.:

   ```
   f :: [[Nat1]] -> [Nat1]
   ```

   such that:

   ```
                         f . g ->> [1,2,6,20,70,...
   ```

3. Implement a transformer `t` dividing the elements of the stream of numbers yielded by (the composition of `g` and) `f` elementwise by the elements of the stream of natural numbers (beginning with 1) as a one-ary co-recursive Haskell function, i.e.:

   ```
   t :: [Nat1] -> [Nat1]
   ```

   such that:

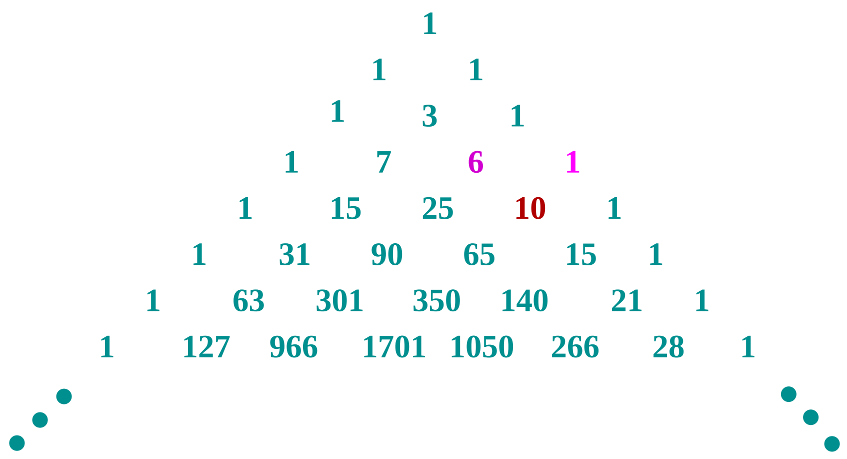
   ```
                  t . (f . g) ->> [1,1,2,5,14,42,132,...
   ```

4. **Without submission:** Implement a variety of selectors of your choice yielding finite parts, sections, or elements of the streams yielded by `g`, `f`, and `t` and their compositions to test and validate your implementations of `g`, `f`, and `t`.

The *Stirling numbers* (of the second kind) $S(n,k)$ are relevant for many counting problems. E.g., they give the number of ways a set S of n elements can be partitioned into k pairwise disjoint non-empty subsets, whose union equals S. For $S =_{df} \{a,b,c\}$, e.g., we get:

$$\begin{array}{lcll} S(3,1) & = & 1 & // \ \{\,\{a,b,c\}\,\} \\ S(3,2) & = & 3 & // \ \{\,\{\{a,b\},\{c\}\},\{\{a\},\{b,c\}\},\{\{a,c\},\{b\}\}\,\} \\ S(3,3) & = & 1 & // \ \{\,\{a\},\{b\},\{c\}\,\} \end{array}$$

Like the binomial coefficients, which are nicely arranged in triangle shape in the Pascal triangle, also the Stirling numbers can nicely be arranged in triangle shape:



The Stirling Triangle

Informally, an entry of the Stirling triangle can be computed using the below (informally stated) recurrence relation, which applies to rows from index 3 onwards:

- A number of the triangle in row i, $i \geq 3$, evolves from the two numbers sitting immediately on top of it by adding to the left of these two numbers the product of the right of the two numbers and the column number for the new Stirling number to be computed.

For an example, consider number 10 located at row 5, column 4 of the prefix of the Stirling triangle shown above. On top of this entry in row 4 sit numbers 6 (to the left) and 1 (to the right). The value of $6 + 1 * 4$ equals 10, which is the value of the entry in row 5, column 4.

Formally, the Stirling numbers can be computed using the recurrence relation:

$$S(n+1,k) = S(n,k-1) + k * S(n,k) \quad \text{for } n = 2,3,\ldots;\ k = 2,\ldots,n$$

5. Implement a Haskell function `s` computing the Stirling numbers using the above recurrence relation:

```
s :: Nat1 -> Nat1 -> Nat1
```

in very much the same way as function `b`:

```
type Nat0 = Integer
b:: Nat0 -> Nat0 -> Nat0
b n k
 | k==0 || n==k = 1
 | True          = b (n-1) k + b (n-1) (k-1)
```

computes the binomial coefficients $(0 \leq k \leq n)$ referring to the recurrence relation:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

6. Implement two (more efficient) variants `s_mhs` and `s_mem` of `s`:

   6.1 `s_mhs :: Nat1 -> Nat1 -> Nat1` (Münchhausen style)

   6.2 `s_mem :: Nat1 -> Nat1 -> Nat1` (Memoization style)

   computing the Stirling numbers utilizing stream programming and a selector together with (6.1) the Münchhausen principle (generalizing the idea underlying the Münchhausen style computation of the stream of Fibonacci numbers to a stream of streams of Stirling numbers (cf. Chapter 2.3.2)), and (6.2) memoization (generalizing the idea of referring to a (1-dimensional) memo list storing computed Fibonacci numbers to a (2-dimensional) memo table storing computed Stirling numbers (cf. Chapter 2.3.3)), respectively.

7. **Without submission:** Compare pairwise the run-time performances of `s`, `s_mh`, and `s_mem` for arguments of growing size. Can you observe performance differences between the three functions? Are they significant?

8. **Without submission:** Can you adapt and use the abstract data type for tables of Chapter 8.5 for your implementations in exercises 6.1 and 6.2? If so, reimplement `s_mhs` and `s_mem` as `s_mhs`′ and `s_mem`′ based on this idea. Compare the four functions regarding conceptual and pragmatical ease of implementation and performance.

Important:

- **Change password:** You should have received by email sent to your generic e-mail address `e<matrikelnummer>@student.tuwien.ac.at` your login data for the computer `g0.complang.tuwien.ac.at` on 12 March 2021. Please, log in as soon as possible (e.g., via `ssh`) and change your inital password to a password of your own choice.

- **Submitting assignments:** Note that your programs will (semi-automatically) be tested and evaluated on `g0` using the there installed GHC interpreter of GHC version 8.65. If you use a different tool or a different version of GHC for program development, please, double-check well in time before the submission deadline that your programs behave on `g0` and the there installed GHC interpreter version as you expect.

Iucundi acti labores.
Getane Arbeiten sind angenehm.

Cicero (106 - 43 v.Chr.)
röm. Staatsmann und Schriftsteller