Advanced Functional Programming: Assignment 2 (Thur, 03/21/2019)
Topic: Streams, Generators, Selectors, and Combinations thereof
Submission deadline: Wed, 04/10/2019 (3pm) (three weeks!)

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

Store all functions to be written for this assignment in a top-level file `assignment2.hs` of your group directory. Comment your program meaningfully; use auxiliary functions and constants, where reasonable.

Co-recursive Definitions of the Stream of Prime Numbers of increasing Performance.

The definition `primes`:

```
primes = sieve [2..]
sieve (p : ns) = p : sieve [ n | n <- ns, mod n p > 0 ]
```

of the stream of prime numbers is usually considered the standard definition in the sense of *The Sieve (of Prime Numbers) of Eratosthenes*. Due to its reliance on `sieve`, the definition of `primes` is indirectly co-recursive. It is famous for its conciseness and elegance but infamous for its poor performance.

In fact, the co-recursive definition `primes_stfwd` (`stwfd` *straightforward*) of the stream of prime numbers, which limits the test of divisibility of new prime number candidates `n` to (already computed) prime numbers up to the size $\sqrt{\texttt{n}}$, performs much better:

```
primes_stfwd = 2 : [ n | n <- [3..], isprime n]
isprime n = all (\p -> mod n p > 0) (primefactorsToTry n)
 where
  primefactorsToTry n = takeWhile (\p -> p*p <= n) primes_stfwd
```

The simple optimization `primes_opt` of `primes`, which does not submit all but only odd natural numbers ≥ 3 for sieving, leads already to a noticeable improvement of the performance, however, without reaching the one of `primes_stfwd`, in particular, as the very same optimization idea can be applied to `primes_stfwd`, too, yielding `primes_stfwd_opt`:

```
primes_opt = 2 : sieve [3,5..]
primes_stfwd_opt = ...
```

1. Implement `primes`, `primes_stfwd`, `primes_opt`, and `primes_stfwd_opt` as shown above, and compare (without submission!) their relative performance.

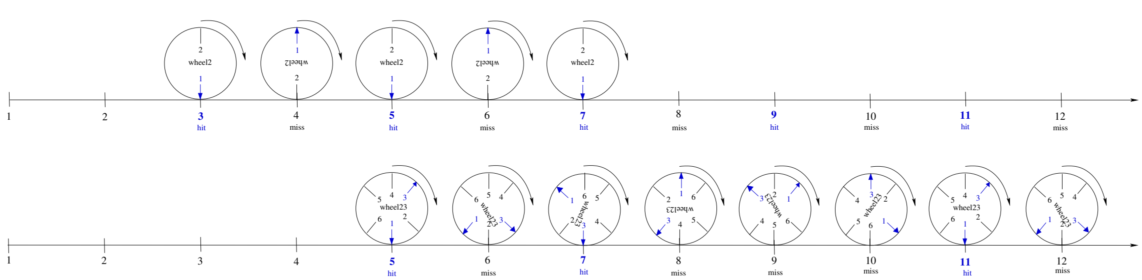
In the following, we focus on further experimenting and practicing computing with streams. To this end, we develop further more and more performant co-recursive definitions of the stream of prime numbers, while accepting that the achieved performance gains can not decisively improve on the asymptotically poor behavior.

`primes_opt` achieves its performance improvement by replacing the stream of natural numbers starting from 2 (`[2..]`) as the stream of prime number candidates by the stream of the odd natural numbers starting from 3 (`[3,5..]`) and by explicitly extracting 2 as a prime number. Intuitively, `primes_opt` halves the count of candidates which must be testet for divisibility, compared to `primes`. In the words of *Baron von Münchhausen*: `primes` pulls itself by grabbing the *empty tuft* (*leeren Schopf*) out of the *swamp* (*Sumpf*) `[2..]`, while `primes_opt` does so by grabbing the one-element *tuft* 2 and pulling itself out of the partially drained *swamp* `[3,5..]` (cf. Chapter 2.1 regarding *tuft* and *swamp*, in particular the co-rexursive tuft/swamp definition of the stream of Fibonacci numbers with tuft `0:1:[]` and the sum of itself and its remainder as swamp).

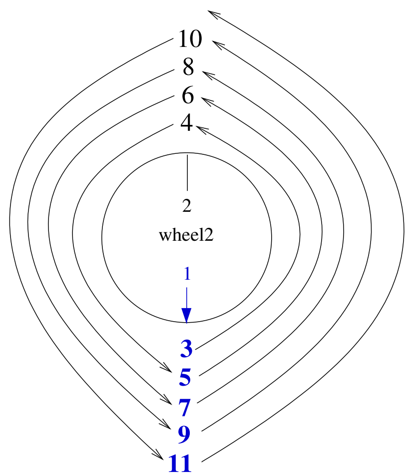
It suggests itself to achieve further performance improvements by successively extending the *tuft*, while simultaneously draining the *swamp*; to swamps, where not only the even numbers, i.e., the multiples of 2 are missing but the multiples of 2 and 3, the multiples of 2, 3, and 5, and so on. Intuitively:

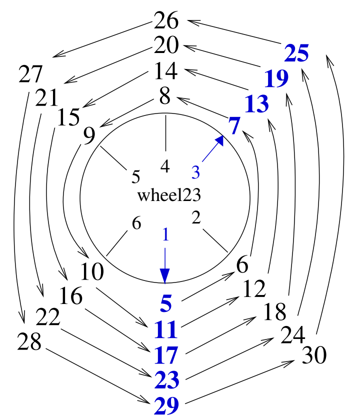
```
primes  = sieve <Stream of nat. numb. from 2>       (= sieve [2..])
primes2 = 2 : sieve <Stream of nat. numb. from 3 w/out multiples of 2>
                                                  (= 2 : sieve [3,5..])
primes23 = 2 : 3 : sieve <Stream of nat. numb. from 5 w/out multiples of 2,3>
primes235 = 2 : 3 : 5 : sieve <Stream of nat. numb. from 7 w/out multiples of 2,3,5>
primes2357 = ...
primes235711 = ...
```

Unlike the (swamp) streams `<Stream of nat. Numb. from 2>` and `<Stream of nat. numb. from 3 w/out multiples of 2>` represented by the Haskell expressions `[2..]` and `[3,5..]`, respectively, we can not describe the other more and more drained (swamp) streams similarly easily in terms of Haskell expressions. However, we can systematically construct them by. To this end, think of a wheel with spikes on its rim rolling along the stream of natural numbers; only numbers which are hit by a spike will be kept as elements of the swamp:



Note that rolling wheel2 yields the stream of numbers [3,5..] = <Stream of nat. numb. of 3 w/out multiples of 2>, that of wheel23 the stream of numbers [5,7,11,13,17,19,23,25,29,31,...] = <Stream of nat. numb. from 5 w/out multiples of 2 and 3>. The below two figures illustrate this differently but equivalently in a dual fashion, where the stream of numbers is spinned around the wheel in the shape of a spiral instead of rolling the wheel along the stream of numbers. Again, this is illustrated for wheel2 and wheel23:





Calling the function spin with wheel resp. wheel2 as swamp generators, and 2 resp. 3 as swamp tufts, the function spin accomplishes the desired; it generates the (swamp) streams <Stream of nat. numb. from 2> and <Stream of nat. numb. from 3 w/out multiples of 2>:

```
wheel = 1 : wheel
wheel2 = 2 : wheel2
spin (x:xs) n = n : spin xs (n+x)

wheel  ->> [1..]
wheel2 ->> [2..]
spin wheel 2  ->> [2..]
spin wheel2 3 ->> [3,5..]
```

Together, this enables the co-recursive definitions primes_wheel and primes_wheel2 of the stream of prime numbers, which could replace the original definitions of primes and primes_opt equivalently:

```
primes_wheel  = sieve (spin wheel 2)        (->> sieve [2..])
primes_wheel2 = 2 : sieve (spin wheel2 3)   (->> sieve [3,5..])

primes     = primes_wheel
primes_opt = primes_wheel2
```

Next, we extend the idea of rolling wheels along the stream of natural numbers to wheels of increasing circumferences: wheel2 has circumference 2 (= 1∗2) and hence 2 positions, where a spike can be or not , `wheel23` has circumference $2*3$ $(= 1*2*3)$ and hence 6 positions, where a spike can be or not, `wheel235` has circumference $2*3*5$ $(= 1*2*3*5)$ and hence 30 positions, where a spike can be or not, etc.; `wheel` as a special case can be thought of as of circumference 1 and hence having 1 position, where a spike can be or not (and actually a spike is).

As seen already, `primes_wheel` and `primes_wheel2` match the definitions of `primes` and `primes_opt` but unlike as `primes` and `primes_opt` can systematically be extended (using function `spin`) to definitions of the stream of prime numbers for wheels of increasing circumferences:

```
wheel = 1 : wheel
wheel2 = 2 : wheel2
wheel23 = <tuft> : wheel23
wheel235 = <tuft> : wheel235
wheel2357 = <tuft> : wheel2357
wheel235711 = <tuft> : wheel235711

spin (x:xs) n = n : spin xs (n+x)

primes_wheel  = sieve (spin wheel 2)       (Tuft empty, swamp origin 2.
                                            Makes swamp [2..])
primes_wheel2 = 2 : sieve (spin wheel2 3)  (Tuft 2, swamp origin 3.
                                            Makes swamp [3,5..])

primes_wheel23     = <tuft> : sieve (spin wheel23 <swamp origin>)
primes_wheel235    = <tuft> : sieve (spin wheel235 <swamp origin>)
primes_wheel2357   = <tuft> : sieve (spin wheel2357 <swamp origin>)
primes_wheel235711 = <tuft> : sieve (spin wheel235711 <swamp origin>)
```

Complete the co-recursive definitions of the

2. wheels `wheel23`, `wheel235`, `wheel2357`, and `wheel235711`, i.e., find the appropriate tufts such that the multiples of 2, 3, of 2, 3, 5, of 2, 3, 5, 7, of 2, 3, 5, 7, 11, respectively, are missed when the wheels are rolled along the stream of natural numbers.

3. streams of prime numbers `primes_wheel23`, `primes_ wheel235`, `primes_wheel2357`, and `primes_wheel235711` induced by the respective wheels, i.e., find the appropriate missing tufts and swamp origins, such that the sieving taking place in the various definitions is applied to the swamps:

   ```
   <Stream of nat. numb. from 5 w/out multiples of 2 and 3>
   <Stream of nat. numb. from 7 w/out multiples of 2, 3 and 5>
   <Stream of nat. numb. from 11 w/out multiples of 2, 3, 5 and 7>
   <Stream of nat. numb. from 13 w/out multiples of 2, 3, 5, 7 and 11>
   ```

4. Compute the stream `tufts` of the tufts of the infinite stream of wheels `wheel`, `wheel2`, `wheel23`, `wheel235`, `wheel2357`, etc.

```
type Nat1       = Integer
type Wheel_Tuft = [Nat1]

tufts :: [Wheel_Tuft]

tufts ->> [[1],[2],...
```

5. Using `tufts`, write a function `wheels` such that:

```
wheels 0 == wheel
wheels 1 == wheel2
wheels 2 == wheel23
wheels 3 == wheel235
wheels 4 == wheel2357
```
`wheels` n `==` `wheel235...p`, `p` nth prime number

6. Using `tufts` or/and `wheels`, write a stream function primes_tailored_wheel, such that the following equalities hold:

```
primes_tailored_wheel 0 == primes_wheel
primes_tailored_wheel 1 == primes_wheel2
primes_tailored_wheel 2 == primes_wheel23
primes_tailored_wheel 3 == primes_wheel235
primes_tailored_wheel 4 == primes_wheel2357
```
`primes_tailored_wheel` n `==` `primes_wheel235...p`, `p` nth prime number

7. **Without submission:**
   - Test all definitions of the stream of prime numbers for functional correctness and compare their relative performances, also with the ones of `primes_stfwd` and `primes_stfwd_opt`.
   - Obviously, the marginal benefit of the wheel-based optimization idea decreases: Every second natural number is a multiple of 2, only every third a multiple of 3, only every fifth a multiple of 5, etc. Moreover, many multiples of 3 are also multiples of 2, many multiples of 5 are also multiples of 2 or/and 3, etc. I.e.,, the tufts of the definitions `primes_wheel...` increase prime number by prime number, the achieved further drain of the swamps, however, gets slower and slower.
     - Can you confirm the decrease of the additional performance gains when observing and comparing the performance gains of your implementations of `primes_wheel...` and `primes_tailored_wheel`, respectively?
     - Can you roughly quantify the respective performance gains from `primes_wheel` to `primes_wheel2` to `primes_wheel23` etc. resp. from `primes_tailored_wheel n` to `primes_tailored_wheel n+1` for some `n` by factors?

- What is the reason that `primes` performs so poorly, that the performance gain delivered by `primes_wheel...` resp. `primes_tailored_wheel` is overall moderate (and decreasing) for increasing wheel sizes? Where is efficiency lost?
- `primes`, `primes_wheel...`, `primes_tailored_wheel` yield unquestionable faithfully the result of the *Sieve of Eratosthenes*. Does this also hold for the concrete way of operationalization? Does it also faithfully mimic the approach of the *Sieve of Eratosthenes*? Could the reason for the loss of efficiency be hidden here?

Important: *Do not use self-defined modules!* If you want to re-use functions (written for earlier assignments), copy these functions to the new submission file. An `import` declaration for self-defined modules will fail, since only the submission file `assignment`i`.hs` , where i, $1 \leq i \leq 8$ (*tentatively*), denotes the running number of the assignment, willl be copied for the (semi-automatic) evaluation. No other file in addition to `assignment`i`.hs` will be copied.