**Advanced Functional Programming: Assignment 8 (Mon, 06/10/2019)**
**Topic: Automatic Program Testing with QuickCheck**
**Submission deadline: Wed, 06/19/2019 (3pm)**

*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 `assignment8.hs` of your group directory. Comment your program meaningfully; use auxiliary functions and constants, where reasonable.

1. Consider the standard implementation `fib` of the Fibonacci function, its significantly more efficient stream based implementation `fibfast`, and the faulty implementation `fibfaulty`:

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

fibstream :: [Int]
fibstream = 0 : 1 : zipWith (+) fibstream (tail fibstream)

fibfast :: Int -> Int
fibfast n = fibstream!!n

fibfaulty :: Int -> Int
fibfaulty n
 | n == 0  = 0
 | n == 1  = 1
 | n <= 10 = fibfaulty (n-1) + fibfaulty (n-2)
 | True    = fibfaulty (n-1) + fibfaulty (n-3)
```

1.1 Define properties for testing, if `fibfast` and `fibfaulty` can be considered faithful implementation variants of the standard implementation `fib` of the Fibonacci function. Properties with postfix

- `-A` shall do this in the most straightforward way offered by QuickCheck, i.e., every integer generated by QuickCheck shall be used as a test input.
- `-B` shall consider only non-negative integers as test inputs using a precondition for filtering test case candidates generated by QuickCheck appropriately.
- `-C` shall use a generator making sure that only non-negative integers are generated as test inputs by QuickCheck.

```
prop_fib_fibfast_A :: Int -> Bool
prop_fib_fibfast_A n = ...
```

```
prop_fib_fibfast_B :: Int -> Property
prop_fib_fibfast_B n = ...
prop_fib_fibfast_C :: Int -> Property
prop_fib_fibfast_C n = ...

prop_fib_fibfaulty_A :: Int -> Bool
prop_fib_fibfaulty_A n = ...
prop_fib_fibfaulty_B :: Int -> Property
prop_fib_fibfaulty_B n = ...
prop_fib_fibfaulty_C :: Int -> Property
prop_fib_fibfaulty_C n = ...
```

1.2 The implementation of `fib` has exponential time complexity. For large(r) test cases the property checks thus take an unduly amount of time. To cope with this, define two new properties `prop_fib_fibfast_B2` and `prop_fib_fibfast_C2`. The implementations of these properties shall ensure that only non-negative values smaller or equal 20 are used as test inputs. Refining the implementations of `prop_fib_fibfast_B` and `prop_fib_fibfast_C`, respectively, the implementations of `prop_fib_fibfast_B2` and `prop_fib_fibfast_C2` shall achieve this using a more sophisticated precondition for test case filtering and a more sophisticaed generator, respectively.

```
prop_fib_fibfast_B2 :: Int -> Property
prop_fib_fibfast_B2 n = ...
prop_fib_fibfast_C2 :: Int -> Property
prop_fib_fibfast_C2 n = ...
```

1.5 Using the QuickCheck combinators `trivial`, `classify`, and `collect`, respectively, extend the implementations of `prop_fib_fibfast_B2` and `prop_fib_fibfast_C2` to get more detailed and informative reports. Using

- `trivial`, `prop_fib_fibfast_B2_trivial` and `prop_fib_fibfast_C2_trivial` shall report the percentage of *trivial* test inputs. As *trivial* we consider the test inputs 0 and 1. A possible report could thus be:

  ```
  OK, passed 100 tests (37% trivial).
  ```

- `classify`, `prop_fib_fibfast_B2_classify` and `prop_fib_fibfast_C2_classify` shall report the percentages of test inputs in the ranges $0 \leq \text{test input} \leq 1$, $2 \leq \text{test input} \leq 10$, and $11 \leq \text{test input}$. A possible report could thus be:

  ```
  OK, passed 100 tests.
  42% of test inputs in the range [0..1].
  37% of test inputs in the range [2..10].
  21% of test inputs in the range [11..].
  ```

- `collect`, `prop_fib_fibfast_B2_collect` and `prop_fib_fibfast_C2_collect` shall report the percentages of all test inputs, i.e., the histogram of test inputs. An excerpt of a possible report could thus be:

```
OK, passed 100 tests.
24% 0.
15% 1.
12% 3.
16% 4.
...
11% 20.
```

2. Integer stacks can be implemented in terms of lists:

```
type Stack  = [Int]
empty       = []
is_empty [] = True
is_empty _  = False
push x xs   = (x:xs)
pop []      = error "Stack is empty"
pop (_:xs)  = xs
top []      = error "Stack is empty"
top (x:_)   = x
```

The above implementation is a correct implementation of integer stacks iff the operations satisfy the laws (a),...,(f):

```
(a) is_empty empty      == True
(b) is_empty (push v s) == False
(c) top empty           == undefined
(d) top (push v s)      == v
(e) pop empty           == undefined
(f) pop (push v s)      == s
```

Obviously, the implementations of `top` and `pop` satisfy law (c) and (e), respectively. Implement properties `prop_a`, `prop_b`, `prop_d`, and `prop_f` allowing to test that the operations in charge also obey the laws (a), (b), (d), and (f):

```
prop_a :: Bool
prop_a = ...
prop_b :: Int -> Stack -> Property
prop_b n ns = ...
prop_d :: Int -> Stack -> Property
prop_d n ns = ...
prop_f :: Int -> Stack -> Property
prop_f n ns = ...
```

Self-defined generators for integer or stack values are not required but differently detailed reports. Property `prop_a` shall just deliver the default report, whereas reports generated by

- `prop_b` shall indicate the percentage of trivial test inputs. A test input is considered trivial, if it involves the empty stack or a singleton stack. A report could thus be:

  ```
  OK, passed 100 tests (24% trivial).
  ```

- `prop_d` shall indicate the percentages of test inputs involving the empty stack, singleton stacks, stacks of size 2, and stacks with more than two entries. A report could thus be:

  ```
  OK, passed 100 tests.
  37% of test inputs: the empty stack.
  28% of test inputs: a singleton stack.
  12% of test inputs: a stack of size 2.
  23% of test inputs: a large stack.
  ```

- `prop_f` shall yield a histogram of the sizes of the stacks involved in the test inputs. A report could thus be:

  ```
  OK, passed 100 tests.
  34% 0.
  25% 1.
  18% 2.
  12% 4.
  11% 6.
  ```

3. Consider the following two implementations of stacks with totally defined pop and top operations. Note that the second component of a `Stack2` value exposes the top element of the stack, if there is one.

```
newtype Stack1 a              = Stk1 (Maybe [a]) deriving (Eq,Show)
is_valid1 (Stk1 (Just _))     = True
is_valid1 (Stk1 Nothing)      = False
empty1                        = Stk1 (Just [])
is_empty1 (Stk1 (Just []))    = True
is_empty1 _                   = False
push1 x (Stk1 (Just xs))      = Stk1 (Just (x:xs))
pust1 x (Stk1 Nothing)        = Stk1 (Just [x])
pop1 (Stk1 (Just []))         = Stk1 Nothing
pop1 (Stk1 (Just (_:xs)))     = Stk1 (Just xs)
pop1 (Stk1 Nothing)           = Stk1 Nothing
top1 :: (Eq a,Show a) => Stack1 a -> Maybe a
top1 (Stk1 (Just []))         = Nothing
top1 (Stk1 (Just (x:_)))      = Just x
top1 (Stk1 Nothing)           = Nothing

newtype Stack2 a              = Stk2 ([a],Maybe a) deriving (Eq,Show)
is_valid2 (Stk2 ([],Nothing)) = True
is_valid2 (Stk2 (_,Nothing))  = False
```

```
is_valid2 (Stk2 ([],Just x))  = False
is_valid2 (Stk2 (xs,Just x))
 | head xs == x               = True
 | True                       = False
empty2                        = Stk2 ([],Nothing)
is_empty2 (Stk2 ([],Nothing)) = True
is_empty2 _                   = False
push2 x (Stk2 (xs,_)          = Stk2 (x:xs,Just x)
pop2 (Stk2 ([],_))            = Stk2 ([],Nothing)
pop2 (Stk2 ([x:[],_))         = Stk2 ([],Nothing)
pop2 (Stk2 ([x:xs,_))         = Stk2 (xs,Just (head xs))
top2 :: (Eq a,Show a) => Stack2 a -> Maybe a
top2 (Stk2 (_,x))             = x
```

3.1 The above two stack implementations are correct iff their operations satisfy the laws (a),...,(f) with 'undefined' replaced by `Nothing`. Implement properties allowing to check this for some of the laws. To this end, make `Stack1` and `Stack2` instances of the required type (constructor) classes of QuickCheck, and implement generators for `(Stack1 Int)` and `(Stack2 Int)` values.

```
prop_stk1_b :: Int -> Stack1 Int -> Property
prop_stk1_b n stk1 = ...   -- checks law b
prop_stk1_d :: Int -> Stack1 Int -> Property
prop_stk1_d n stk1 = ...   -- checks law d
prop_stk1_f :: Int -> Stack1 Int -> Property
prop_stk1_f n stk1 = ...   -- checks law f

prop_stk2_b :: Int -> Stack2 Int -> Property
prop_stk2_b n stk2 = ...  -- checks law b
prop_stk2_d :: Int -> Stack2 Int -> Property
prop_stk2_d n stk1 = ...   -- checks law d
prop_stk2_f :: Int -> Stack2 Int -> Property
prop_stk2_f n stk1 = ...   -- checks law f
```

3.2 The functions `retrieve1` and `retrieve2` link `Stack1` and `Stack2` values:

```
retrieve1 :: (Stack1 a) -> (Stack2 a)
retrieve1 (Stk1 Nothing)   = Stk2 ([],Nothing)
retrieve1 (Stk1 Just [])   = Stk2 ([],Nothing)
retrieve1 (Stk1 (Just xs)) = Stk2 (xs,Just (head xs))

retrieve2 :: (Stack2 a) -> (Stack1 a)
retrieve2 (Stk2 ([],Nothing)  = Stk1 (Just [])
retrieve2 (Stk2 ([],_)        = Stk1 Nothing
retrieve2 (Stk2 (xs,Just x))
   | head xs == x = Stk1 (Just xs)
   | True         = Stk1 Nothing
```

```
retrieve2 (Stk2 (xs,Nothing)) = Stk1 Nothing
```

Implement properties allowing to check that both implementations can mutually be considered implementation variants of each other. To this end implement properties for tessting this partially, again for `Int` stacks. Reusing the generators of Exercise 3.1, complete the below property definitions where necessary, and add type signatures for them:

```
prop_isvalid1 stk1 = forall <insert generator> $
                      is_valid1 stk1 == is_valid2 (retrieve1 stk1)
prop_empty1 = retrieve1 empty1 == empty2
prop_push1 n stk1 = forall <insert generator> $
                     retrieve1 (push1 n stk1) == push2 n (retrieve1 stk1)
prop_pop1 stk1 = forall <insert generator> $
                  retrieve1 (pop1 stk1) == pop2 (retrieve1 stk1)
prop_top1 stk1 = forall <insert generator> $
                  top1 stk1 == top2 (retrieve1 stk1)
prop_isvalid_pop1 stk1 = forall <insert generator> $
                          is_valid2 (retrieve1 (pop1 stk1)) ==
                            is_valid2 (pop2 (retrieve1 stk1))

prop_isvalid2 stk2 = forall <insert generator> $
                      is_valid2 stk2 == is_valid1 (retrieve2 stk2)
prop_empty2 = retrieve2 empty2 == empty1
prop_push2 n stk2 = forall <insert generator> $
                     retrieve2 (push2 n stk2) == push1 n (retrieve2 stk2)
prop_pop2 stk2 = forall <insert generator> $
                  retrieve2 (pop2 stk2) == pop1 (retrieve2 stk2)
prop_top2 stk2 = forall <insert generator> $
                  top2 stk2 == top1 (retrieve2 stk2)
prop_isvalid_pop2 stk2 = forall <insert generator> $
                          is_valid1 (retrieve2 (pop2 stk2)) ==
                            is_valid1 (pop1 (retrieve2 stk2))
```

3.3 **Without submission:** Are there properties of Exercise 3.1 and 3.2 which can be falsified? If so, what are the reasons for this? Faulty/sloppy implementations of stack operations? Faulty/sloppy implementations of the retrieve functions? Faulty/sloppy generator implementations, which can generate non-wellformed stack values? Other reasons? Can possible faults be fixed such that all properties can successfully be checked?

3.4 **Without submission:** Develop variants of the property definitions of Excercise 3.2 which provide more detailed information on the kind of stack values used as test inputs.

**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.