Functional programming in Haskell
Second exercise sheet

Sebastian Muskalla

TU Braunschweig

Summer term 2018

Out: June 5

The exercises will be discussed during the exercise class on Thursday, June 7, 9:45 in IZ 305.

The code snippets from this sheet are available as .hs file on the lecture page.

Exercise 1: Exception handling via monads

Design a type constructor `Exception` such that a value of type `Exception a` is either an exception with an error message (of type `String`) or a result of type `a`.

Implement the necessary functions to make `Exception` a `Monad`.

Exercise 2: From non-monadic to monadic code

Consider the following code that specifies a data type for simple arithmetic expressions involving constants and division, and an evaluation function.

```
1 data Arith = Const Integer | Div Arith Arith deriving Show
2
3 eval :: Arith → Integer
4 eval (Const n) = n
5 eval (Div l r) = (eval l) `div` (eval r)
6
7 good, bad :: Arith
8 good = Div (Div (Const 1932) (Const 2)) (Const 23)
9 bad  = Div (Const 1) (Const 0)
```

a) Reimplement function `eval` such that it uses the trivial monad.

b) Extend the code from Part a) such that dividing by zero gives an exception. Use the `Exception` monad from Exercise 1.

c) Extend the code from Part a) such that it counts the number of divisions. Use the monad `State Int`.

d) Extend the code from Part a) such that it counts prints all intermediary results. Use the `IO` monad.

How many modifications to the code do you need to make for the Parts b) to d)?

Exercise 3: Parsing bracket-free expressions

a) Consider the following code that specifies a data type for arithmetic expressions.

```
data Expression =
        Constant Integer
    |   Add Expression Expression
    |   Subtract Expression Expression
    |   Multiply Expression Expression
    |   Divide Expression Expression

testexpr = Add (Constant 2) (Multiply (Constant 3) (
   Subtract (Constant 4) (Divide (Constant 2) (Constant 3)
   )))
```

Implement the instance Show Expression with a function show that returns the fully bracketed, infix representation of an expression, e.g.
`show testexpr == "(2 + (3 * (4 - (2 / 3))))".`

b) Consider the following code that specifies again a data type for arithmetic expressions.

```
data BExpression = T Term | A Term Term | S Term Term
       -- recursion, addition, subtraction
data Term  = F Factor | M Factor Factor | D Factor Factor
    -- recursion, multiplication, division
data Factor      = C Integer | B BExpression
                -- constant, expression in brackets

testbexpr =
  A (F (C 2)) (M (C 3) (B (S (F (C 4)) (D (C 2) (C 3)))))
```

Implement the instance Show BExpression with a function show that returns the infix representation of an expression with as few brackets as possible e.g.
`show testbexpr == "2 + 3 * (4 - 2 / 3)".`
As usual, brackets bind stronger then · and /, which in turn bind stronger than + and –. Operators of the same precedence are processed from left to right.

c) Implement a function `convert :: BExpression -> Expression`.

d) Implement a function `parser :: Parser BExpression` that can parse `String` representations of `BExpressions` back into `BExpression`

Note: The lecture material contains a Parser for `Expression` from Part a) that may serve as a basis.

Hint: A naive implement may run into an infinite loop because of the so-called left-recursion problem. To avoid this, use the following transformed grammar:

```
bexpr ::= term (addop term)*
 term ::= factor (multop factor)*
factor ::= Integer | "(" bexpr ")"
addop ::= "+" | "-"
multop ::= "*" | "/"
```

Exercise 4: State and ST

a) Read about the `State` monad, e.g. at `https://wiki.haskell.org/State_Monad`

b) Read about the ST (state-thread) monad, e.g. at `https://wiki.haskell.org/Monad/ST`

c) Implement your favorite algorithm in an imperative, non-pure fashion using the ST monad.

Note: The lecture material contains implementations of the Fibonacci function and of the quicksort algorithm, both taken from Richard Bird's book.