ITMO FP Course Homework Checker Download

Homework #0

Task 1

1. Create a module named `HW0.T1` and define the following type in it:

```
data a <-> b = Iso (a -> b) (b -> a)

flipIso :: (a <-> b) -> (b <-> a)
flipIso (Iso f g) = Iso g f

runIso :: (a <-> b) -> (a -> b)
runIso (Iso f _) = f
```

2. Implement the following functions and isomorphisms:

```
distrib :: Either a (b, c) -> (Either a b, Either a c)
assocPair :: (a, (b, c)) <-> ((a, b), c)
assocEither :: Either a (Either b c) <-> Either (Either a b) c
```

Task 2

1. Create a module named `HW0.T2` and define the following type in it:

```
type Not a = a -> Void
```

2. Implement the following functions and isomorphisms:

```
doubleNeg :: a -> Not (Not a)
reduceTripleNeg :: Not (Not (Not a)) -> Not a
```

Task 3

1. Create a module named `HW0.T3` and define the following combinators in it:

```
s :: (a -> b -> c) -> (a -> b) -> (a -> c)
s f g x = f x (g x)

k :: a -> b -> a
k x y = x
```

2. Using *only those combinators* and function application (i.e. no lambdas, pattern matching, and so on) define the following additional combinators:

```
i :: a -> a
compose :: (b -> c) -> (a -> b) -> (a -> c)
contract :: (a -> a -> b) -> (a -> b)
permute :: (a -> b -> c) -> (b -> a -> c)
```

   For example:

```
i x = x         -- No (parameters on the LHS disallowed)
i = \x -> x     -- No (lambdas disallowed)
i = Prelude.id  -- No (only use s and k)
i = s k k       -- OK
i = (s k) k     -- OK (parentheses for grouping allowed)
```

Task 4

1. Create a module named `HW0.T4`.

2. Using the `fix` combinator from the `Data.Function` module define the following functions:

```
repeat' :: a -> [a]             -- behaves like Data.List.repeat
map' :: (a -> b) -> [a] -> [b]  -- behaves like Data.List.map
fib :: Natural -> Natural       -- computes the n-th Fibonacci number
fac :: Natural -> Natural       -- computes the factorial
```

   Do not use explicit recursion. For example:

```
repeat' = Data.List.repeat     -- No (obviously)
repeat' x = x : repeat' x      -- No (explicit recursion disallowed)
repeat' x = fix (x:)           -- OK
```

Task 5

1. Create a module named `HW0.T5` and define the following type in it:

```
type Nat a = (a -> a) -> a -> a
```

2. Implement the following functions:

```
nz :: Nat a
ns :: Nat a -> Nat a

nplus, nmult :: Nat a -> Nat a -> Nat a

nFromNatural :: Natural -> Nat a
nToNum :: Num a => Nat a -> a
```

3. The following equations must hold:

```
nToNum nz       ==  0
nToNum (ns x)   ==  1 + nToNum x

nToNum (nplus a b)   ==   nToNum a + nToNum b
nToNum (nmult a b)   ==   nToNum a * nToNum b
```

Task 6

1. Create a module named `HW0.T6` and define the following values in it:

```
a = distrib (Left ("AB" ++ "CD" ++ "EF"))     -- distrib from HW0.T1
b = map isSpace "Hello, World"
c = if 1 > 0 || error "X" then "Y" else "Z"
```

2. Determine the WHNF (weak head normal form) of these values:

```
a_whnf = ...
b_whnf = ...
c_whnf = ...
```

Homework #1

Task 1

1. Create a module named `HW1.T1` and define the following data type in it:

```
data Day = Monday | Tuesday | ... | Sunday
```

   (Obviously, fill in the `...` with the rest of the week days).

   Do not derive `Enum` for `Day`, as the derived `toEnum` is partial:

```
ghci> toEnum 42 :: Day
*** Exception: toEnum{Day}: tag (42) is outside of enumeration's range (0,6)
```

2. Implement the following functions:

```
-- | Returns the day that follows the day of the week given as input.
nextDay :: Day -> Day

-- | Returns the day of the week after a given number of days has passed.
afterDays :: Natural -> Day -> Day

-- | Checks if the day is on the weekend.
isWeekend :: Day -> Bool

-- | Computes the number of days until Friday.
daysToParty :: Day -> Natural
```

In `daysToParty`, if it is already `Friday`, the party can start immediately, we don't have to wait for the next week (i.e. return `0` rather than `7`).

Good job if you spotted that this task is a perfect fit for modular arithmetic, but that is not the point of the exercise. The functions must be implemented by operating on `Day` values directly, without conversion to a numeric representation.

Task 2

1. Create a module named `HW1.T2` and define the following data type in it:

```
data N = Z | S N
```

2. Implement the following functions:

```
nplus :: N -> N -> N        -- addition
nmult :: N -> N -> N        -- multiplication
nsub :: N -> N -> Maybe N   -- subtraction     (Nothing if result is negative)
ncmp :: N -> N -> Ordering  -- comparison      (Do not derive Ord)
```

The operations must be implemented without using built-in numbers (`Int`, `Integer`, `Natural`, and such).

3. Implement the following functions:

```
nFromNatural :: Natural -> N
nToNum :: Num a => N -> a
```

4. (Advanced) Implement the following functions:

```
nEven, nOdd :: N -> Bool    -- parity checking
ndiv :: N -> N -> N         -- integer division
nmod :: N -> N -> N         -- modulo operation
```

The operations must be implemented without using built-in numbers.

In `ndiv` and `nmod`, the behavior in case of division by zero is not specified (you can throw an exception, go into an inifnite loop, or delete all files on the computer).

Task 3

1. Create a module named `HW1.T3` and define the following data type in it:

```
data Tree a = Leaf | Branch Meta (Tree a) a (Tree a)
```

The `Meta` field must store additional information about the subtree that can be accessed in constant time, for example its size. You can use `Int`, `(Int, Int)`, or a custom data structure:

```
type Meta = Int         -- OK
data Meta = M Int Int   -- OK
```

Functions operating on this tree must maintain the following invariants:

1. **Sorted**: The elements in the left subtree are less than the head element of a branch, and the elements in the right subtree are greater.
2. **Unique**: There are no duplicate elements in the tree (follows from **Sorted**).
3. **CachedSize**: The size of the tree is cached in the `Meta` field for constant-time access.
4. (Advanced) **Balanced**: The tree is balanced according to one of the following strategies:
   - **SizeBalanced**: For any given `Branch _ l _ r`, the ratio between the size of `l` and the size of `r` never exceeds `3`.
   - **HeightBalanced**: For any given `Branch _ l _ r`, the difference between the height of `l` and the height of `r` never exceeds `1`.

These invariants enable efficient processing of the tree.

2. Implement the following functions:

```
-- | Size of the tree, O(1).
tsize :: Tree a -> Int

-- | Depth of the tree.
tdepth :: Tree a -> Int

-- | Check if the element is in the tree, O(log n)
tmember :: Ord a => a -> Tree a -> Bool

-- | Insert an element into the tree, O(log n)
tinsert :: Ord a => a -> Tree a -> Tree a

-- | Build a tree from a list, O(n log n)
tFromList :: Ord a => [a] -> Tree a
```

Tip 1: in order to maintain the **CachedSize** invariant, define a helper function:

```
mkBranch :: Tree a -> a -> Tree a -> Tree a
```

Tip 2: the **Balanced** invariant is the hardest to maintain, so implement it last. Search for “tree rotation”.

Task 4

1. Create a module named `HW1.T4`.

2. Using the `Tree` data type from `HW1.T3`, define the following function:

```
tfoldr :: (a -> b -> b) -> b -> Tree a -> b
```

It must collect the elements in order:

```
treeToList :: Tree a -> [a]    -- output list is sorted
treeToList = tfoldr (:) []
```

This follows from the **Sorted** invariant.

You are encouraged to define `tfoldr` in an efficient manner, doing only a single pass over the tree and without constructing intermediate lists.

Task 5

1. Create a module named `HW1.T5`.

2. Implement the following function:

```
splitOn :: Eq a => a -> [a] -> NonEmpty [a]
```

Conceptually, it splits a list into sublists by a separator:

```
ghci> splitOn '/' "path/to/file"
["path", "to", "file"]

ghci> splitOn '/' "path/with/trailing/slash/"
["path", "with", "trailing", "slash", ""]
```

Due to the use of `NonEmpty` to enforce that there is at least one sublist in the output, the actual GHCi result will look slightly differently:

```
ghci> splitOn '/' "path/to/file"
"path" :| ["to","file"]
```

Do not let that confuse you. The first element is not in any way special.

3. Implement the following function:

```
joinWith :: a -> NonEmpty [a] -> [a]
```

It must be the inverse of `splitOn`, so that:

```
(joinWith sep . splitOn sep)  ≡  id
```

Example usage:

```
ghci> "import " ++ joinWith '.' ("Data" :| "List" : "NonEmpty" : [])
"import Data.List.NonEmpty"
```

Task 6

1. Create a module named `HW1.T6`.

2. Using `Foldable` methods *only*, implement the following function:

```
mcat :: Monoid a => [Maybe a] -> a
```

Example usage:

```
ghci> mcat [Just "mo", Nothing, Nothing, Just "no", Just "id"]
"monoid"

ghci> Data.Monoid.getSum $ mcat [Nothing, Just 2, Nothing, Just 40]
42
```

3. Using `foldMap` to consume the list, implement the following function:

```
epart :: (Monoid a, Monoid b) => [Either a b] -> (a, b)
```

Example usage:

```
ghci> epart [Left (Sum 3), Right [1,2,3], Left (Sum 5), Right [4,5]]
(Sum {getSum = 8},[1,2,3,4,5])
```

Task 7

1. Create a module named `HW1.T7`.

2. Define the following data type and a lawful `Semigroup` instance for it:

```
data ListPlus a = a :+ ListPlus a | Last a
infixr 5 :+
```

3. Define the following data type and a lawful `Semigroup` instance for it:

   ```
   data Inclusive a b = This a | That b | Both a b
   ```

   The instance must not discard any values:

   ```
   This i  <>  This j  =  This (i <> j)   -- OK
   This i  <>  This _  =  This i          -- This is not the Semigroup you're looking for.
   ```

4. Define the following data type:

   ```
   newtype DotString = DS String
   ```

   Implement a `Semigroup` instance for it, such that the strings are concatenated with a dot:

   ```
   ghci> DS "person" <> DS "address" <> DS "city"
   DS "person.address.city"
   ```

   Implement a `Monoid` instance for it using `DS ""` as the identity element. Make sure that the laws hold:

   ```
   mempty <> a  ≡  a
   a <> mempty  ≡  a
   ```

5. Define the following data type:

   ```
   newtype Fun a = F (a -> a)
   ```

   Implement lawful `Semigroup` and `Monoid` instances for it.

HOMEWORK #2

Task 1

1. Create a module named `HW2.T1` and define the following data types in it:

   - ```
     data Option a = None | Some a
     ```
   - ```
     data Pair a = P a a
     ```
   - ```
     data Quad a = Q a a a a
     ```
   - ```
     data Annotated e a = a :# e
     infix 0 :#
     ```
   - ```
     data Except e a = Error e | Success a
     ```
   - ```
     data Prioritised a = Low a | Medium a | High a
     ```
   - ```
     data Stream a = a :> Stream a
     infixr 5 :>
     ```
   - ```
     data List a = Nil | a :. List a
     infixr 5 :.
     ```
   - ```
     data Fun i a = F (i -> a)
     ```
   - ```
     data Tree a = Leaf | Branch (Tree a) a (Tree a)
     ```

2. For each of those types, implement a function of the following form:

   ```
   mapF :: (a -> b) -> (F a -> F b)
   ```

   That is, implement the following functions:

   ```
   mapOption      :: (a -> b) -> (Option a -> Option b)
   mapPair        :: (a -> b) -> (Pair a -> Pair b)
   ```

```
mapQuad        :: (a -> b) -> (Quad a -> Quad b)
mapAnnotated   :: (a -> b) -> (Annotated e a -> Annotated e b)
mapExcept      :: (a -> b) -> (Except e a -> Except e b)
mapPrioritised :: (a -> b) -> (Prioritised a -> Prioritised b)
mapStream      :: (a -> b) -> (Stream a -> Stream b)
mapList        :: (a -> b) -> (List a -> List b)
mapFun         :: (a -> b) -> (Fun i a -> Fun i b)
mapTree        :: (a -> b) -> (Tree a -> Tree b)
```

These functions must modify only the elements and preserve the overall structure (e.g. do not reverse the list, do not rebalance the tree, do not swap the pair).

This property is witnessed by the following laws:

```
          mapF id  ≡  id
mapF f ∘ mapF g   ≡  mapF (f ∘ g)
```

You must implement these functions by hand, without using any predefined functions (not even from `Prelude`) or deriving.

Task 2

Create a module named `HW2.T2`. For each type from the first task except `Tree`, implement functions of the following form:

```
distF :: (F a, F b) -> F (a, b)
wrapF :: a -> F a
```

That is, implement the following functions:

```
distOption      :: (Option a, Option b) -> Option (a, b)
distPair        :: (Pair a, Pair b) -> Pair (a, b)
distQuad        :: (Quad a, Quad b) -> Quad (a, b)
distAnnotated   :: Semigroup e => (Annotated e a, Annotated e b) -> Annotated e (a, b)
distExcept      :: (Except e a, Except e b) -> Except e (a, b)
distPrioritised :: (Prioritised a, Prioritised b) -> Prioritised (a, b)
distStream      :: (Stream a, Stream b) -> Stream (a, b)
distList        :: (List a, List b) -> List (a, b)
distFun         :: (Fun i a, Fun i b) -> Fun i (a, b)
```

```
wrapOption      :: a -> Option a
wrapPair        :: a -> Pair a
wrapQuad        :: a -> Quad a
wrapAnnotated   :: Monoid e => a -> Annotated e a
wrapExcept      :: a -> Except e a
wrapPrioritised :: a -> Prioritised a
wrapStream      :: a -> Stream a
wrapList        :: a -> List a
wrapFun         :: a -> Fun i a
```

The following laws must hold:

- Homomorphism:

  ```
  distF (wrapF a, wrapF b)  ≅  wrapF (a, b)
  ```

- Associativity:

  ```
  distF (p, distF (q, r))   ≅  distF (distF (p, q), r)
  ```

- Left and right identity:

  ```
  distF (wrapF (), q)  ≅  q
  distF (p, wrapF ())  ≅  p
  ```

In the laws stated above, we reason up to the following isomorphisms:

```
((a, b), c)  ≅  (a, (b, c))  -- for associativity
    ((), b)  ≅  b            -- for left identity
    (a, ())  ≅  a            -- for right identity
```

There is more than one way to implement some of these functions. In addition to the laws, take the following expectations into account:

- `distPrioritised` must pick the higher priority out of the two.
- `distList` must associate each element of the first list with each element of the second list (i.e. the resulting list is of length `n × m`).

You must implement these functions by hand, using only:

- data types that you defined in `HW2.T1`
- `(<>)` and `mempty` for `Annotated`

Task 3

Create a module named `HW2.T3`. For `Option`, `Except`, `Annotated`, `List`, and `Fun` define a function of the following form:

```
joinF :: F (F a) -> F a
```

That is, implement the following functions:

```
joinOption    :: Option (Option a) -> Option a
joinExcept    :: Except e (Except e a) -> Except e a
joinAnnotated :: Semigroup e => Annotated e (Annotated e a) -> Annotated e a
joinList      :: List (List a) -> List a
joinFun       :: Fun i (Fun i a) -> Fun i a
```

The following laws must hold:

- Associativity:

  ```
  joinF (mapF joinF m)  ≡  joinF (joinF m)
  ```

  In other words, given `F (F (F a))`, it does not matter whether we join the outer layers or the inner layers first.

- Left and right identity:

  ```
  joinF      (wrapF m)  ≡  m
  joinF (mapF wrapF m)  ≡  m
  ```

  In other words, layers created by `wrapF` are identity elements to `joinF`.

  Given `F a`, you can add layers outside/inside to get `F (F a)`, but `joinF` flattens it back into `F a` without any other changes to the structure.

Furthermore, `joinF` is strictly more powerful than `distF` and can be used to define it:

```
distF (p, q) = joinF (mapF (\a -> mapF (\b -> (a, b)) q) p)
```

At the same time, this is only one of the possible `distF` definitions (e.g. `List` admits at least two lawful `distF`). It is common in Haskell to expect `distF` and `joinF` to agree in behavior, so the above equation must hold. (Do not redefine `distF` using `joinF`, though: it would be correct but not the point of the exercise).

Task 4

1. Create a module named `HW2.T4` and define the following data type in it:

   ```
   data State s a = S { runS :: s -> Annotated s a }
   ```

2. Implement the following functions:

```
mapState :: (a -> b) -> State s a -> State s b
wrapState :: a -> State s a
joinState :: State s (State s a) -> State s a
modifyState :: (s -> s) -> State s ()
```

   Using those functions, define `Functor`, `Applicative`, and `Monad` instances:

```
instance Functor (State s) where
  fmap = mapState

instance Applicative (State s) where
  pure = wrapState
  p <*> q = Control.Monad.ap p q

instance Monad (State s) where
  m >>= f = joinState (fmap f m)
```

   These instances will enable the use of `do`-notation with `State`.

   The semantics of `State` are such that the following holds:

```
runS (do modifyState f; modifyState g; return a) x
  ≡
a :# g (f x)
```

   In other words, we execute stateful actions left-to-right, passing the state from one to another.

3. Define the following data type, representing a small language:

```
data Prim a =
    Add a a      -- (+)
  | Sub a a      -- (-)
  | Mul a a      -- (*)
  | Div a a      -- (/)
  | Abs a        -- abs
  | Sgn a        -- signum

data Expr = Val Double | Op (Prim Expr)
```

   For notational convenience, define the following instances:

```
instance Num Expr where
  x + y = Op (Add x y)
  x * y = Op (Mul x y)
  ...
  fromInteger x = Val (fromInteger x)

instance Fractional Expr where
  ...
```

   So that `(3.14 + 1.618 :: Expr)` produces this syntax tree:

```
Op (Add (Val 3.14) (Val 1.618))
```

4. Using `do`-notation for `State` and combinators we defined for it (`pure`, `modifyState`), define the evaluation function:

```
eval :: Expr -> State [Prim Double] Double
```

   In addition to the final result of evaluating an expression, it accumulates a trace of all individual operations:

```
runS (eval (2 + 3 * 5 - 7)) []
  ≡
10 :# [Sub 17 7, Add 2 15, Mul 3 5]
```

The head of the list is the last operation, this way adding another operation to the trace is O(1).

You can use the trace to observe the evaluation order. Consider this expression:

```
(a * b) + (x * y)
```

In `eval`, we choose to evaluate `(a * b)` first and `(x * y)` second, even though the opposite is also possible and would not affect the final result of the computation.

Task 5

1. Create a module named `HW2.T5` and define the following data type in it:

   ```
   data ExceptState e s a = ES { runES :: s -> Except e (Annotated s a) }
   ```

   This type is a combination of `Except` and `State`, allowing a stateful computation to abort with an error.

2. Implement the following functions:

   ```
   mapExceptState :: (a -> b) -> ExceptState e s a -> ExceptState e s b
   wrapExceptState :: a -> ExceptState e s a
   joinExceptState :: ExceptState e s (ExceptState e s a) -> ExceptState e s a
   modifyExceptState :: (s -> s) -> ExceptState e s ()
   throwExceptState :: e -> ExceptState e s a
   ```

   Using those functions, define `Functor`, `Applicative`, and `Monad` instances.

3. Using `do`-notation for `ExceptState` and combinators we defined for it (`pure`, `modifyExceptState`, `throwExceptState`), define the evaluation function:

   ```
   data EvaluationError = DivideByZero
   eval :: Expr -> ExceptState EvaluationError [Prim Double] Double
   ```

   It works just as `eval` from the previous task but aborts the computation if division by zero occurs:

   - ```
     runES (eval (2 + 3 * 5 - 7)) []
       ≡
     Success (10 :# [Sub 17 7, Add 2 15, Mul 3 5])
     ```
   - ```
     runES (eval (1 / (10 - 5 * 2))) []
       ≡
     Error DivideByZero
     ```

Task 6

1. Create a module named `HW2.T6` and define the following data type in it:

   ```
   data ParseError = ErrorAtPos Natural

   newtype Parser a = P (ExceptState ParseError (Natural, String) a)
     deriving newtype (Functor, Applicative, Monad)
   ```

   Here we use `ExceptState` for an entirely different purpose: to parse data from a string. Our state consists of a `Natural` representing how many characters we have already consumed (for error messages) and the `String` is the remainder of the input.

2. Implement the following function:

   ```
   runP :: Parser a -> String -> Except ParseError a
   ```

3. Let us define a parser that consumes a single character:

```
pChar :: Parser Char
pChar = P $ ES \(pos, s) ->
  case s of
    []     -> Error (ErrorAtPos pos)
    (c:cs) -> Success (c :# (pos + 1, cs))
```

Study this definition:

- What happens when the string is empty?
- How does the parser state change when a character is consumed?

Write a comment that explains `pChar`.

4. Implement a parser that always fails:

```
parseError :: Parser a
```

Define the following instance:

```
instance Alternative Parser where
  empty = parseError
  (<|>) = ...

instance MonadPlus Parser   -- No methods.
```

So that `p <|> q` tries to parse the input string using `p`, but in case of failure tries `q`.

Make sure that the laws hold:

```
empty <|> p  ≡  p
p <|> empty  ≡  p
```

5. Implement a parser that checks that there is no unconsumed input left (i.e. the string in the parser state is empty), and fails otherwise:

```
pEof :: Parser ()
```

6. Study the combinators provided by `Control.Applicative` and `Control.Monad`. The following are of particular interest:

- `msum`
- `mfilter`
- `optional`
- `many`
- `some`
- `void`

We can use them to construct more interesting parsers. For instance, here is a parser that accepts only non-empty sequences of uppercase letters:

```
pAbbr :: Parser String
pAbbr = do
  abbr <- some (mfilter Data.Char.isUpper pChar)
  pEof
  pure abbr
```

It can be used as follows:

```
ghci> runP pAbbr "HTML"
Success "HTML"

ghci> runP pAbbr "JavaScript"
Error (ErrorAtPos 1)
```

7. Using parser combinators, define the following function:

```
parseExpr :: String -> Except ParseError Expr
```

It must handle floating-point literals of the form `4.09`, the operators `+ - * /` with the usual precedence (multiplication and division bind tighter than addition and subtraction), and parentheses.

Example usage:

- ```
  ghci> parseExpr "3.14 + 1.618 * 2"
  Success (Op (Add (Val 3.14) (Op (Mul (Val 1.618) (Val 2.0)))))
  ```
- ```
  ghci> parseExpr "2 * (1 + 3)"
  Success (Op (Mul (Val 2.0) (Op (Add (Val 1.0) (Val 3.0)))))
  ```
- ```
  ghci> parseExpr "24 + Hello"
  Error (ErrorAtPos 3)
  ```

The implementation must not use the `Read` class, as it implements similar functionality (the exercise is to write your own parsers). At the same time, you are encouraged to use existing `Applicative` and `Monad` combinators, since they are not specific to parsing.
  ```