Take Home Exam

When and How

The exam should be handed in using the **lab reporting system** by 17:00 on Wednesday, August 27.

Solutions

There are several possible solutions to both problems. Here are the ones I had in mind when constructing the exam.

- **Problem 1a** [**code**]
- **Problem 1b** [**code**]
- **Problem 1c** [**code**]
- **Problem 2b** [**code**]
- **Problem 2c** [**code**]
- **Problem 2d** [**code**]

Important

You are not allowed to discuss the exam problems with anyone. You are free to study the course material or other resources to figure out your solutions, but **the code you write must be your own**.

To pass the exam you need to show a basic understanding of monads and embedded languages in Haskell, by providing reasonably good attempts at solutions to the problems below.

If anything is unclear, feel free to **send me a mail** with your question.

Problem 1

Consider the following class

```
class Monad m => GameMonad m where
  extraLife  :: m ()
  getLives   :: m Int
  checkPoint :: m a -> m a
  die        :: m a
```

When computing in a game monad you have a number of lives. The current number of lives is returned by `getLives` and executing `extraLife` increases the number of lives by one. When you die you lose one life, and if you have any lives remaining you get to start over from the closest enclosing `checkPoint`.

Hints: Consider an execution of `checkPoint m` for some computation `m`. There are three cases:

- `m` runs to completion, possibly gaining or losing lives in the process. The computation succeeds with the result of `m`.
- `m` dies, but there are still lives remaining so `m` is executed again.
- `m` dies and there are no more lives, in which case the computation fails.

Your implementation of a game monad needs to be able to capture all three possibilies.

Some examples of game monad computations:

```
printLives :: (GameMonad m, MonadIO m) => String -> m ()
printLives s = do
  n <- getLives
  liftIO $ putStrLn $ s ++ " " ++ show n

example1 :: (GameMonad m, MonadIO m) => m ()
example1 = checkPoint $ do
  printLives "Lives:"
  die
  liftIO $ putStrLn "We win!"

-- Only succeeds if we're on the last life.
lastChance :: GameMonad m => m ()
lastChance = do
```

```
    n <- getLives
    if n == 1 then return ()
              else die

  example2 :: (GameMonad m, MonadIO m) => m String
  example2 = checkPoint $ do
    printLives "Start"
    n <- getLives
    if n == 1
      then do
        liftIO $ putStrLn "Finish"
        return "Victory!"
      else do
        checkPoint $ do
          printLives "Inner checkpoint"
          lastChance
        extraLife
        printLives "Extra life!"
        die
```

When running the examples using the run function from **(b)** the following happens:

```
ghci> runGameT example1 3
Lives: 3
Lives: 2
Lives: 1
Nothing
ghci> runGameT example2 3
Start 3
Inner checkpoint 3
Inner checkpoint 2
Inner checkpoint 1
Extra life! 2
Start 1
Finish
Just ("Victory!",1)
```

(a) Implement a game monad by defining a datatype Game and giving instances for the Monad and GameMonad classes. You are not allowed to use the library monads from `Control.Monad.*` when defining Game. You should also define a run function

```
runGame :: Game a -> Int -> Maybe (a, Int)
```

The second argument to runGame is the number of lives you start with. If the game finishes without running out of lives runGame should return `Just (x, n)`, where x is the result of the computation and n is the remaining number of lives.

(b) Implement a game monad transformer GameT. Besides the Monad and GameMonad instances, you should also give an instance of the **MonadTrans** class. Once again you are not allowed to use the library monads. To be able to run the examples above you can use the following instance of **MonadIO**:

```
instance MonadIO m => MonadIO (GameT m) where
  liftIO = lift . liftIO
```

Also define the run function

```
runGameT :: Monad m => GameT m a -> Int -> m (Maybe (a, Int))
```

with the same behaviour as runGame above.

(c) Now define a version of GameT using the monads from `Control.Monad.*`. Your implementation of GameT should have the form

```
type GameT m = SomeMonadT args (AnotherMonadT args ..)
```

In this case you get the Monad and MonadTrans instances for free, so you only have to give the GameMonad instance. To be allowed to define this instance you need to give the flag -XTypeSynonymInstances to ghc or add the following line at the top of your file.

```
{-# LANGUAGE TypeSynonymInstances #-}
```

The run function should have the same type as in **(b)**.

Problem 2

Implement an embedded language for pretty printing. It should support the following operations:

```
type Doc
($$)   :: Doc -> Doc -> Doc -- vertical composition
(<>)   :: Doc -> Doc -> Doc -- horizontal composition (no separation)
(<+>)  :: Doc -> Doc -> Doc -- horizontal composition (separated by a space)
empty  :: Doc               -- an empty document
text   :: String -> Doc     -- a document containing the given string
indent :: Int -> Doc -> Doc -- adds a given number of spaces to each line in a document
render :: Doc -> String     -- render a document as a string
```

You don't have to consider the efficiency of the library. Some examples:

```
ghci> putStr $ render $ text "xxx" <+> text "yyy"
xxx yyy
ghci> putStr $ render $ text "A" $$ empty $$ text "B"
A
B
ghci> putStr $ render $ (text "xxxxx" $$ text "xx") <> (text "yy" $$ text "yyy")
xxxxx
xxyy
  yyy
ghci> putStr $ render $ (text "a" $$ text "bc") <> indent 2 (text "XX" $$ text "YY")
a
bc  XX
    YY
```

Note that horizontal composition adds the second document to the last line of the first document, as can be seen in the third example above.

(a) Identify any derived operations, i.e. operations that can be expressed in terms of other operations, and give their definitions.

(b) Give a shallow embedding of the language.

(c) Give a deep embedding of the language.

(d) *(Bonus problem)* Add a function cat and change the type of render

```
cat    :: Doc -> Doc -> Doc
render :: Int -> Doc -> String
```

where the first argument to render is the desired width (the length of the longest line) of the document, and cat acts as horizontal composition unless that would make the document to wide and vertical composition otherwise.