Kiel University
Department of Computer Science
Programming Languages and Compiler Construction

Priv.-Doz. Dr. Frank Huch, Sandra Dylus, B.Sc. Jan-Hendrik Matthes, Rajib Chandra Das

Institut für Informatik

5. Exam on „Advanced Programming (pre-masters)“

SS 19

You can obtain 28 points within two assignments. To pass the test, you have to reach at least 14 points.

For the test, you can work for 90 minutes. It is not allowed to use any material other than a pen. Mobile phones have to be turned off.

Hold your Student ID Card ready, we will check it during the examination.

You will be informed about the results on Friday, September 20th, 2019 from 2:00PM to 3:00PM in CAP4 - R.715.

Exercise 1 - Multiple Choice and Questions

12 Points

Multiple Choice

Mark all correct answers with an X. **If you want to change your answer after marking a statement, fill the original square and draw a new one, as shown in the example.** Note that any number of answers can be correct. Each question is worth 1.5 points. For each incorrect answer 0.5 points are deducted; negative scores for a question are not possible.

Example

1. Which cities are in Germany?
   - ☒ Kiel
   - ☒ Hamburg
   - ☐ London
   - ■ ☐ Paris
   - ☒ Berlin

Questions

1. Which of the following expressions cannot be typed in Haskell (i.e., yield a type error)?
   - ☐ `Just Nothing`
   - ☐ `(1,2,3.0)`
   - ☐ `getLine >>= return`
   - ☐ `foldr (+) 0`
   - ☐ `[Left 3, Right True]`
2. Which of the following functions reverses a list?
   - ☐ `foldl (:) []`
   - ☐ `foldr (:) []`
   - ☐ `foldl (++) []`
   - ☐ `foldr (++) []`
   - ☐ `foldr (\x acc -> acc ++ [x]) []`
3. Which of the following predefined functions are higher-order?
   - ☐ `map`
   - ☐ `foldl`
   - ☐ `zip`
   - ☐ `reverse`
   - ☐ `filter`
4. Which of the following expressions evaluate to 42?
   - ☐ `foldr (+) 6 [5,11,20]`
   - ☐ `uncurry (+) (20,22)`
   - ☐ `foldl (*) 3 [1,2,7]`
   - ☐ `[1..100] !! 42`
   - ☐ `fst (6 + 30, 42)`

5. Which of the following types are valid?

   ☐ `filter :: (a -> Bool) -> [a] -> [Bool]`

   ☐ `foldr (+) :: Int -> [Int] -> [Int]`

   ☐ `foldr (\ _ _ -> 42) :: a -> [a] -> Int`

   ☐ `fst (map, 42) :: (a -> b) -> [a] -> [b]`

   ☐ `zip [1,2,3] :: [a] -> [Int]`

Answer the following questions directly on this sheet of paper.

1. *(1.5 points)* We define the following algebraic data types.

```
data One   where
  One :: One
data Rec a b where
  Const :: a -> Rec a b
  Rec   :: Rec a b -> b -> Rec a b
```

   Give three different values of type `Rec One One`.

2. *(1.5 points)* Define a mapping function for `Rec a b` that implements the following type signature.

```
mapRec :: (a -> c) -> Rec a b -> Rec c b
```

3. *(2 points)* Give the most general type signature for the following function.

```
f ::
f g h x = h (g x x)
```

Exercise 2 - Functions, Datatypes and IO

16 Points

Write your solutions for the following exercises directly on this sheet of paper.

1) **(2 Points)** Define a polymorphic data type `NonEmptyList` that represents non-empty lists with polymorphic elements.

```
data NonEmptyList    where
```

2) **(2 Points)** Define a function `inputInRange :: Int -> Int -> Int -> InRange` that checks if the third parameter is greater than the first and smaller than the second argument. The result type `InRange` is defined as follows.

```
data InRange    where
  Below  :: InRange
  Above  :: InRange
  Within :: InRange
 deriving Show

inputInRange :: Int -> Int -> Int -> InRange
inputInRange
```

3) **(2 Points)** The following function `getIntInRange :: Int -> Int -> IO Int` asks the user to enter a number that is within the range of the first (lower bound) and second (upper bound) argument until it's valid. If the user's input is not in range of these two bounds, a message should be printed indicating if the input was too great or too small. Rewrite the function `getIntInRange` without using do-notation.

```
getIntInRange :: Int -> Int -> IO Int
getIntInRange lower upper = do
  str <- getLine
  if all isDigit str
    then let n = read str
         in case inputInRange lower upper (read str) of
              Within  -> return n
              _ -> do
                putStrLn ("The␣number␣is␣too␣high␣or␣low.")
                getIntInRange lower upper
    else do
      putStrLn "Invalid␣input;␣please␣type␣in␣a␣number."
      getIntInRange lower upper

getIntInRangeNoDo :: Int -> Int -> IO Int
getIntInRangeNoDo lower upper =
```

4) **(3 Points)** Define a function `displayMenu :: [String] -> IO ()` that prints all entries of the given list as an enumerated menu for the user. Your implementation should produce the following output for the exemplary list `["Hangman", "Guess a Number", "Rock, Paper, Scissors"]`.

```
> displayMenu ["Hangman", "Guess a Number", "Rock, Paper, Scissors"]
(1) Hangman
(2) Guess a Number
(3) Rock, Paper, Scissors
```

```
displayMenu :: [String] -> IO ()
displayMenu
```

5) **(5 Points)** Use the functon `getIntInRange` to define the function `menu :: [(String, IO GameResult)] -> IO ()` that implements and interactive menu to play different games that are passed as first argument. That is, the first component of the pair represents the name of the game and second component represents the game itself. In order for the user to make a choice, the names of the games should be presented using

`displayMenu`. The user's input then determines which game to play (depending on the position of the game entry in the list). Valid inputs for the example menu above are `1`, `2` and `3`. If the user makes a valid choice, the corresponding game should be executed, the result of the game should then be presented to the user with an adequate message. An exemplary message to congratulate user `A` for winning looks as follows.

```
Congratulations, Player A won the game!
```

```
data TwoPlayer    where
  A :: TwoPlayer
  B :: TwoPlayer
 deriving Show

data GameResult    where
  Win  :: TwoPlayer -> GameResult
  Lose :: TwoPlayer -> GameResult
  Draw :: GameResult
 deriving Show

menu :: [(String, IO GameResult)] -> IO ()
menu games = do
  putStrLn "Which␣game␣do␣you␣want␣to␣play?"
```

6) **(2 Points)** The above `menu` function can only handle games that use `TwoPlayer` as representation for players. Generalise the data type `GameResult` such it can also be used with different representations.

```
data GameResult    where
  Win  ::
  Lose ::
  Draw ::
```