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

2. 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 present, we will check it during the examination.

You will be informed about the results on Monday, June 24, 2019 from 3PM to 4PM in HRS3 - R.105a/R.105b.

Exercise 1 - Quiz

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 will yield a type error?
   - ☐ `map (\x y -> x * x + y) [1, 2, 3]`
   - ☐ `Just Nothing`
   - ☐ `foldl (++) [] [1 : 2 : [], [3]]`
   - ☐ `[(1, 2), (1, 2, 3)]`
   - ☐ `([42], False, True)`
2. Which of the following types are correct?
   - ☐ `putStrLn "Hello, world!" :: IO ()`
   - ☐ `map :: (a -> a) -> [a] -> [b]`
   - ☐ `foldr :: (a -> b -> b) -> b -> [a] -> b`
   - ☐ `(>>) :: a -> b -> a`
   - ☐ `getLine :: IO String`
3. Which of the following expressions terminate (if they are entered into ghci)?
   - ☐ `zip [1..] [2, 4, 6, 8, 10]`
   - ☐ `take 10 [10..]`
   - ☐ `let loop = loop in head loop`
   - ☐ `let loop = 1 : loop in head loop`
   - ☐ `let loop = 1 : loop in tail loop`
4. Which of the following functions are defined within the type class `Ord`?
   - ☐ `compare`
   - ☐ `gt`
   - ☐ `ordering`
   - ☐ `(>=)`
   - ☐ `(/=)`

5. Which of the following types have only three different values?

   ☐ `(Bool, Bool)`

   ☐ `Either () Bool`

   ☐ `Maybe (Maybe ())`

   ☐ `((), Bool)`

   ☐ `Maybe Bool`

Answer the following questions directly on this sheet of paper.

1. *(2 points)* We define the following algebraic data type for lists.

```
data List a where
  NoElem      :: List a
  OneMoreElem :: a -> List a -> List a
```

   Implement the function `prepend :: List a -> List a -> List a` that combines two lists by prepending the second list in front of the first list. That is, the function `prepend` yields the following exemplary result.

```
> prepend (OneMoreElem 1 (OneMoreElem 2 NoElem)) (OneMoreElem 3 (OneMoreElem 4 NoElem))
OneMoreElem 3 (OneMoreElem 4 (OneMoreElem 1 (OneMoreElem 2 NoElem)))
```

2. *(1.5 points)* Translate the following IO program with `do`-notation into an equivalent program without `do`.

```
main :: IO ()
main = do getChar >>= putChar
          str <- return "Hello!"
          putStrLn (' ':str)
```

3. *(1 point)* We define the following algebraic data types.

```
data Unit where
  Unit :: Unit

data Two a where
  TwoRec  :: Two a -> Unit -> Two a
  TwoPoly :: a -> Two a
```

   Give two different values of type `Two (Two Unit)`.

Exercise 2 - I/O, Arithmetic Expressions and Polymorphic Binary Node-Labeled Trees 16 Points

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

1. *(8 points)* In this exercise you have to implement a simple version of the game **rock-paper-scissors** in Haskell. This is a hand game usually played between two people, in which each player simultaneously forms one of three shapes (rock, paper or scissors) with an outstretched hand. The game has the following three rules.

   - *Rock* beats *scissors*.
   - *Scissors* beats *paper*.
   - *Paper* beats *rock*.

   If both players choose the same shape, the game is tied.

   - *(1 point)* Define a data type `Shape` to represent the three different shapes.
   - *(3 points)* Implement a helper function `getShape :: IO Shape` that asks the user for input until the user types one of the shapes (i.e., `o`, `[]`, or `8<`). For other shapes, the user is notified about the invalid input.
   - *(2 points)* Next, define a function `result :: Shape -> Shape -> Ordering` that compares the shape of the user with the shape of the computer.
   - *(2 points)* Finally, complete the function `help :: IO ()` in the function `play :: IO ()` defined below to implement the main logic of the game. The decision about winning and losing is determined by the return value of `result`. Only a draw results in a rematch, otherwise the game ends. The function `randomShape :: IO Shape` does not need to be defined!

```
1 play :: IO ()
2 play = do putStrLn "Welcome␣to␣Rock-Paper-Scissors!"
3           help
4   where
5     help :: IO ()
6     help = do uShape <- getShape    -- Shape of the user.
7               cShape <- randomShape -- Shape of the computer.
8               -- TODO: Your code here.
```

   Your game output should look as follows.

```
ghci> play
Welcome to Rock-Paper-Scissors!
Rock `o`, Paper `[]` or Scissors `8<`?
> Rock
Invalid input! Please try again.
Rock `o`, Paper `[]` or Scissors `8<`?
> 123
Invalid input! Please try again.
Rock `o`, Paper `[]` or Scissors `8<`?
> []
Draw! Try again!
Rock `o`, Paper `[]` or Scissors `8<`?
> 8<
Won!
```

2. *(2 points)* Define a data type in GADT-syntax to represent an arithmetic expression. An arithmetic expression can be an integer, the sum of two arithmetic expressions or the square of an arithmetic expression.

3. *(6 points)* Consider the following polymorphic data type `BTree a` to represent **node-labeled** binary trees with labels of type `a`.

```
1 data BTree a where
2   Empty :: BTree a
3   Node  :: BTree a -> a -> BTree a -> BTree a
```

Implement the following functions for binary trees. Don't forget to annotate each function with the corresponding type signature!

- *(1 point)* Define a smart constructor `leaf` which constructs a binary tree (more specifically a leaf) from a given label.
- *(3 points)* Define the function `foldBTree` which folds a binary tree into a value of another type. For example, we can fold a binary tree with integer labels to a single integer by using the predefined `(+)` function to sum up all the labels.
- *(2 points)* Finally, define an instance of the type class `Eq` for `BTree a`, which behaves like the implementation you would get with `deriving Eq`.