HTWK Leipzig, Fakultät IM

Prof. Dr. Sibylle Schwarz sibylle.schwarz@htwk-leipzig.de

9.Übung zur Vorlesung „Fortgeschrittene Programmierung“
Sommersemester 2020 zu lösen bis 17. Juni 2020

zur Recherche im Haskell-Standard `https://www.haskell.org/onlinereport/haskell2010`

Aufgabe 9.1 (Typklassen – Syntax)

zum Abschnitt „Type Classes and Overloading“ :
Geben Sie eine Ableitung und einen Ableitungsbaum bezüglich der dort angegebenen Grammatik und der Start-Variablen `topdecl` an für: `class B t => C t where { f :: t -> Bool }`
Dabei wird (wenigstens) eine Grammatik-Variable benutzt, deren Regeln in einem anderen Kapitel definiert sind. Welche? Wo?

Aufgabe 9.2 (Typklassen – Statische Semantik)

Beantworten Sie durch exakten Verweis auf den Standard:
Warum ist das folgende Programm falsch?

```
class B t => C t where { f :: t -> Bool }
class C t => B t where { g :: t -> Bool }
```

Geben Sie die Definition des dabei benutzen mathematischen Modells an. Stellen Sie fest, ob `ghc` die Formulierung aus dem Standard in der Fehlermeldung verwendet.

Aufgabe 9.3 (Typklassen – Statische Semantik)

Warum ist die Deklaration `class C t where f :: Bool` verboten?
Beantworten Sie durch Verweis auf Standard (Hinweis: „. . . must mention u“)
und erklären Sie an der Auswertung von `f` für

```
instance C Bool where f = False
instance C Int  where f = True
```

Aufgabe 9.4 (Gleichheit auf Listen)

Suchen Sie auf `https://hackage.haskell.org/` die tatsächliche Implementierung der Gleichheit auf Listen. Dort steht (wo genau?):

```
instance (Eq a) => Eq [a] where
    []     == []     = True
    (x:xs) == (y:ys) = x == y && xs == ys
    _xs    == _ys    = False
```

Geben Sie für jedes Vorkommen des Operators `==` an:

a. ob der Operator dort definiert oder benutzt wird

b. für welche Instantiierung des Typ-Argumentes der Operator definiert oder benutzt wird.

c. Welche der Benutzungen ist eine Rekursion?

d. Wozu ist das Constraint `Eq a` notwendig?

Aufgabe 9.5 (Verwendung von Default-Implementierungen)

Im Haskell-Standard (wo genau?) steht

```
class  Eq a  where
        (==), (/=)  ::  a -> a -> Bool
        x /= y  = not (x == y)
        x == y  = not (x /= y)
```

Beschreiben Sie die Auswertung von `A == A` für

```
data T = A
instance Eq T where { A /= A = False }
```

Aufgabe 9.6 (Verwendung von Default-Implementierungen)

Im Haskell-Standard steht

```
class  (Eq a) => Ord a  where
    compare              :: a -> a -> Ordering
    (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min             :: a -> a -> a

    compare x y | x == y    = EQ
                | x <= y    = LT
                | otherwise = GT

    x <= y  = compare x y /= GT
    x <  y  = compare x y == LT
    x >= y  = compare x y /= LT
    x >  y  = compare x y == GT

    -- Note that (min x y, max x y) = (x,y) or (y,x)
    max x y | x <= y    =  y
            | otherwise =  x
    min x y | x <= y    =  x
            | otherwise =  y
```

a. Wo steht das genau?

b. Beschreiben Sie die Auswertung von `A > B` für

```
data T = A | B deriving Eq
instance Ord T where { B <= A = False ; _ <= _ = True }
```

c. Warum ist die Default-Implementierung von `min` nicht:

```
min x y | x >= y    =  y
        | otherwise =  x
```

Erläutern Sie an diesem Beispiel:

```
data T = T Bool Bool deriving Eq
instance Ord T where compare (T a b) (T c d) = compare a c
x = T False False ; y = T False True
max x y ; min x y
```

d. In der Implementierung durch die Standardbibliothek des GHC-Compilers steht (wo genau?) bei der Default-Implementierung von `compare` zusätzlich die Bemerkung

```
-- NB: must be '<=' not '<' to ...
```

Nehmen Sie an, man ändert die Default-Implementierung auf

```
compare x y | x == y    = EQ
            | x <  y    = LT
            | otherwise = GT
```

(restlicher Quelltext bleibt). Beschreiben Sie die Auswertung von `A > B` nach

```
data T = A | B deriving Eq
instance Ord T where { B <= A = False ; _ <= _ = True }
```

Aufgabe 9.7 (Abgeleitete Instanzen)

a. Lesen Sie die Spezifikation für abgeleitete (derived) Eq- und Ord-Instanzen im Haskell-Standard.

b. Gegeben ist die folgende Typdeklaration

```
data List a = Cons a (List a) | Nil
  deriving (Eq, Ord)
```

Welche Paare von Konstruktoren werden bei der Auswertung von
`Cons True (Cons False Nil) < Cons False Nil`
in welcher Reihenfolge verglichen?

c. Geben die monoton aufsteigende Ordnung dieser Elemente an:

```
Cons True (Cons False Nil)
Nil
Cons True Nil
Cons False Nil
```

Aufgabe 9.8 (Quasi-lexikografische Ordnung auf Listen)

a. Wiederholen Sie die Definition der quasi-lexikographischen Ordnung.

b. Zeigen Sie durch ein Beispiel, dass die quasi-lexikographische Ordnung verschieden von der lexikographischen Ordnung ist.

c. Geben Sie vier Zahlenfolgen an, für welche die lexikographisch aufsteigende Permuation elementweise verschieden ist von der quasi-lexikographisch aufsteigenden Permutation.

Da es `instance Ord a => Ord [a]` bereits gibt, können wir diese nicht ändern, sondern definieren einen neuen Datentyp. Dazu gibt es eine Autotool-Aufgabe.
Dieses Vorgehen ist softwaretechnisch richtig: eine anwendungsspezifische Kombination von Standard-Typen.
(Die Benutzung von `[a]` ist jedoch meist falsch, siehe
`https://www.imn.htwk-leipzig.de/~waldmann/etc/untutorial/list-or-not-list`)