universität innsbruck

Funktionale Programmierung	WS 2026/2027	LVA 703206/703025
Übungsblatt 4, 10 Punkte	Abgabefrist: Dienstag, 17. November 2026, 20:00 Uhr	

- Kreuzen Sie gelöste Aufgaben im OLAT Kurs des Proseminars an.
- Lösen Sie Programmieraufgaben in Template_04.hs und laden Sie diese Datei in OLAT hoch.
- Ihre Template-Datei sollte mit `ghci` ohne Fehlermeldung kompilieren.
- **Wichtig**: Die Tests mittels `cabal run` kompilieren nur, wenn Sie den Datentyp in Aufgabe 1 definiert haben. Falls Sie Aufgabe 1 nicht bearbeitet haben, müssen Sie die Tests mit Id `"1.*"Tests_04.hs` auskommentieren, genauso die jeweiligen Funktionsdefinitionen.

Aufgabe 1 *Boolesche Ausdücke* **5 P.**

In dieser Aufgabe betrachten Sie unterschiedliche Darstellungen von Booleschen Ausdrücken. In der Rechnerarchitektur sind Gatter, Hardware-Signale und Bits (`Zero`/`One`) üblich, während in der Theoretischen Informatik Boolesche Ausdrücke über abstrakte Variablennamen und Wahrheitswerte (`True`/`False`) ausgewertet werden. Wir verwenden einen polymorphen Datentyp, um Boolesche Ausdrücke mit derselben logischen Struktur, für unterschiedlichen Repräsentationen darstellen zu können.

1. Definieren Sie einen parametrischen Datentyp `BoolExpr var val` für boolesche Ausdrücke. Diese dürfen aus Variablen (`Var`), Werten (`Val`), Negationen (`Not`), Konjunktionen (`And`) und Disjunktionen (`Or`) bestehen. Hierbei soll `var` der Typ der Variablen und `val` der Typ der Werte sein. (1 Punkt)

   Beispiel:
   ```
   formula1 :: BoolExpr String Bool
   formula1 = And (Var "x") (Val True)
   ```

   Beispiel:
   ```
   circuit1 :: BoolExpr Signal Bit
   circuit1 = Or (Not (Var (Signal "enable" 0))) (Val Zero)
   ```

2. Definieren Sie eine Funktion `circuitToFormula`, die eine Schaltung vom Typ `BoolExpr Signal Bit` in eine boolesche Formel vom Typ `BoolExpr String Bool` übersetzt. Dabei sollen Signale mittels der vordefinierten Funktion `signalToVarString` übersetzt werden. Außerdem soll `Zero` durch `False` und `One` durch `True` ersetzt werden. Die Struktur des Ausdrucks soll erhalten bleiben. (2 Punkte)

   Beispiel:
   ```
   circuitToFormula circuit1 == Or (Not (Var "enable_p0")) (Val False)
   ```

3. Definieren Sie eine Funktion `evalFormula`, die eine boolesche Formel und eine Variablen-Belegung in Form von Schlüssel-Wert-Paaren erhält und den Wert der Formel für die gegebene Variablen-Belegung berechnet.

   Falls eine Variable nicht in der Variablen-Belegung vorkommt, soll sie als `False` interpretiert werden. (2 Punkte)

   Beispiel:
   ```
   evalFormula [("x", True), ("y", False)] (And (Var "x") (Not (Var "y"))) == True
   evalFormula [("x", True)] (And (Var "x") (Var "y")) == False
   ```

   Hinweis: Sie önnen die vordefinierte Funktion `lookup :: Eq a => a -> [(a,b)] -> Maybe b` verwenden um den Wert einer Variable nachzuschlagen. Diese liefert `Nothing`, wenn der gesuchte Schlüssel nicht vorkommt.

Aufgabe 2 *Polymorphie und Listen* **2 P.**

In dieser Aufgabe definieren Sie polymorphe Funktionen auf Listenelementen. Die Funktionen sollen für Listen mit beliebigem Elementtyp funktionieren.

1. Definieren Sie eine Funktion `swapPairs :: [(a, b)] -> [(b, a)]`, die in jedem Paar einer Liste die beiden Komponenten vertauscht. (1 Punkt)

   Beispiel:
   ```
   swapPairs [(4,'clubs'),(5,'diamonds')] = [('clubs',4),('diamonds',5)]
   swapPairs [("Red",True)] = [(True,"Red")]
   ```

2. Definieren Sie eine Funktion `interleave :: [a] -> [a] -> [a]`, die zwei Listen abwechselnd zusammenfügt, beginnend mit dem ersten Element der ersten Liste. Ist eine Liste länger als die andere, werden ihre restlichen Elemente am Ende angehängt. (1 Punkt)

   Beispiel:
   ```
   interleave [5,6,7] [9,10] = [5,9,6,10,7]
   interleave "spades" "clubs" = "scplaudbess"
   ```

Aufgabe 3 *Maybe, Either, und Listen* **3 P.**

In dieser Aufgabe definieren Sie Funktionen auf Listen, die mit den Typen `Maybe` und `Either` fehlende oder ungültige Werte darstellen.

1. Die Funktion `head` aus der Prelude wirft eine Exception, wenn sie auf die leere Liste angewendet wird. Definieren Sie eine Funktion `safeHeads :: [[a]] -> [Maybe a]`, die für jede Teilliste einer Liste von Listen deren erstes Element in `Just` verpackt zurückgibt. Für leere Teillisten soll `Nothing` zurückgegeben werden. (1 Punkt)

   Beispiel:
   ```
   safeHeads [[6,7],[],[9]] = [Just 6,Nothing,Just 9]
   safeHeads ["Spades","","Hearts"] = [Just 'S',Nothing,Just 'H']
   ```

2. Definieren Sie eine Funktion `fromMaybes :: a -> [Maybe a] -> [a]`, die einen Standardwert und eine Liste von `Maybe`-Werten erhält. Jedes `Just x` soll durch `x` und jedes `Nothing` durch den Standardwert ersetzt werden. (1 Punkt)

   Beispiel:
   ```
   fromMaybes 0 [Just 6,Nothing,Just 7] = [6,0,7]
   fromMaybes '_' [Just 'H',Nothing,Just 'P'] = "H_P"
   ```

3. Definieren Sie eine Funktion `initials :: [String] -> String`, die aus einer Liste von Wörtern die Anfangsbuchstaben bildet. Für ein leeres Wort soll das Zeichen `'_'` verwendet werden. Verwenden Sie dazu `safeHeads` und `fromMaybes` aus den Teilaufgaben (a) und (b). (1 Punkt)

   Beispiel:
   ```
   initials ["Haskell","","Prolog"] = "H_P"
   initials ["Maybe","Either"] = "ME"
   ```