universität innsbruck

Funktionale Programmierung	WS 2025/2026	LVA 703025
Übungsblatt 12, 10 Punkte		Abgabefrist: Mittwoch, 28. Januar 2026, 6 Uhr

- Kreuzen Sie gelöste Aufgaben im OLAT Kurs des Proseminars an.
- Lösen Sie Programmieraufgaben in `Template_12.hs` und `Stack.hs`, und laden Sie die Dateien separat in OLAT hoch.
- Ihre Dateien sollten mit `ghci` ohne Fehlermeldung kompilieren.

Aufgabe 1 *Zyklische Listen* **3 P.**

Wir nennen eine Zahl n *magisch*, wenn sie eine der folgenden zwei Bedingungen erfüllt:

- $n = 1$, oder
- es gibt eine magische Zahl m, so dass $n = 3m$ oder $n = 7m$ oder $n = 13m$.

Das Ziel dieser Aufgabe ist es, die unendliche Liste aller magischen Zahlen in aufsteigender Reihenfolge zu berechnen.

1. Schreiben Sie eine Funktion `merge`, die zwei Listen zu einer zusammenführt. `merge xs ys` sollte die folgenden Bedingungen erfüllen:
   - Alle Elemente in `merge xs ys` sind auch Elemente in `xs` oder `ys`.
   - Alle Elemente in `xs` oder `ys` sind auch Elemente in `merge xs ys`.
   - Wenn `xs` und `ys` in aufsteigender Reihenfolge sortiert sind und keine Duplikate enthalten, dann ist auch `merge xs ys` aufsteigend sortiert und enthält keine Duplikate.

   Beispiel: `merge [1,5,200] [9,17,200,400] = [1,5,9,17,200,400]` (1 Punkt)

2. Definieren Sie die unendliche Liste `mNumbers`, welche die unendliche Liste der magischen Zahlen in aufsteigender Reihenfolge ohne Duplikate als zyklische Liste berechnet.

   Tipp: Verwenden Sie die Funktion `merge` und Funktionen wie `map (3*)`. Schauen Sie sich auch die Definition von `fibs` auf Folie 7 der Vorlesung 12 an.

   Beispiel: `take 10 mNumbers = [1,3,7,9,13,21,27,39,49,63]` (2 Punkte)

Aufgabe 2 *Abstrakte Datentypen, Stackmaschine* **7 P.**

1. Ein `Stack` ist ein abstrakter Datentyp mit der folgenden Signatur:

   ```
   empty :: Stack a
   isEmpty :: Stack a -> Bool
   size :: Stack a -> Int
   push :: a -> Stack a -> Stack a
   pop :: Stack a -> (a, Stack a)
   ```

   Der Datentyp soll die folgende Spezifikation erfüllen:

   ```
   isEmpty empty
   not (isEmpty (push x s))
   pop (push x s) = (x, s)
   size (push x s) = 1 + size s
   ```

   In der Datei `Stack.hs` finden Sie eine Implementierung eines Stacks, die diese Eigenschaften erfüllt.

- Es fehlen noch einige Eigenschaften. Vervollständigen Sie die Spezifikation.
- Die Implementierung von `size` ist derzeit ineffizient. Modifizieren Sie die Implementierung so, dass `size` eine Operation mit konstanter Laufzeit wird.

(1 Punkt)

2. Wir betrachten eine einfache Programmiersprache P, die auf einem nur-lesenden Speicher und einem Stack basiert. Ein P-Programm ist einfach eine Liste von Maschineninstruktionen (`Instr`). Es gibt vier verschiedene Instruktionen:
   - `Const` x legt die konstante Zahl x auf den Stack.
   - `Load` l liest den Inhalt c der Speicheradresse l und legt c auf den Stack.
   - `Multiply` entfernt zwei Werte vom Stack und legt ihr Produkt wieder auf den Stack.
   - `Subtract` entfernt zwei Werte vom Stack und legt ihre Differenz wieder auf den Stack.

   Das Ausführen eines P-Programms für einen gegebenen Speicher m erfolgt, indem eine Instruktion nach der anderen ausgeführt wird, beginnend mit einem anfangs leeren Stack. Das Ergebnis der Ausführung wird bestimmt, indem ein Wert vom finalen Stack entfernt wird.

   Beispiel: Das Ausführen des Programms
   `[Const 5, Load "x", Const 3, Subtract, Load "y", Multiply]` für einen Speicher, in dem x den Wert 17 und y den Wert 9 speichert, liefert das Ergebnis -126.

   Begründung: Die Maschine legt zunächst 5, dann 17 und dann 3 auf den Stack. Anschließend werden 3 und 17 entfernt und ihre Differenz -14 auf den Stack gelegt. Danach wird 9 aus der Speicherzelle y auf den Stack gelegt. Schließlich werden 9 und -14 entfernt und ihr Produkt -126 auf den Stack gelegt. Am Ende wird -126 entfernt und als Ergebnis zurückgegeben.

   Aufgabe: Schreiben Sie eine Haskell-Funktion
   `runProgram :: (String -> Integer) -> [Instr] -> Integer`, die die Ausführung eines P-Programms simuliert. (3 Punkte)

3. Betrachten Sie einfache arithmetische Ausdrücke.

```
data Expr = Var String | Num Integer | Minus Expr Expr | Times Expr Expr deriving Show

eval a (Num n) = n
eval a (Var x) = a x
eval a (Minus e1 e2) = eval a e1 - eval a e2
eval a (Times e1 e2) = eval a e1 * eval a e2
```

   Schreiben Sie einen Compiler von Ausdrücken in P-Programme, also eine Funktion
   `compile :: Expr -> [Instr]`, so dass die folgende Eigenschaft erfüllt ist. (3 Punkte)

```
runProgram a (compile e) == eval a e
```