Aufgabe 6-1 (Type)

a) Erstellen Sie mit Hilfe von `type` einen Datentyp `Parabel`, der die Koeffizienten einer beliebigen Parabel speichert. Die Koeffizienten sollen Double oder Integer sein können. Dieser Koeffiziententyp soll durch das Ändern einer einzelnen Zeile im Programm angegeben werden können. Implementieren Sie zudem folgende Funktionen:
- `eval` - Berechnet den Wert einer Parabel an einem gegebenen x-Wert
- `derive` - Berechnet die Ableitung einer Parabel
- `slope` - Berechnet die Steigung der Parabel an einem gegebenen x-Wert

Achten Sie darauf, dass alle Eingaben und Ausgaben der Funktionen den, in der ersten Zeile angegebenen, Koeffiziententyp haben.

b) Welche folgende Typdefinitionen sind zulässig bzw. unzulässig und warum?

```
type Line = [String]
type Double = [Int]
type Liste = (Char, Liste)
type Bool = Bool
type words = String
```

Aufgabe 6-2 (Data - Listen)

a) Definieren Sie mit Hilfe von `data` einen eigenen Listentypen `NewList`, der die Funktionsweise einer „normalen“ Haskell-Liste hat.

b) Schreiben Sie für den neuen Listentypen folgende Funktionen (Hinweis: durch Klammerung von nicht-alphabetischen Zeichen können Infix-Funktionen geschrieben werden):

```
-- Anzahl der Elemente
newLength :: NewList a -> Int

-- Entfernt die ersten n Elemente und gibt den Rest zurück
newDrop :: Int -> NewList a -> NewList a

-- Äquivalent zu (:) bei normalen Listen, fügt ein
-- Element an den Anfang der Liste an.
-- z.B. fügt ( 2 # list ) die 2 als erstes
-- Element an "list" an (Infix-Notation)
(#) :: a -> NewList a -> NewList a

-- Gibt die ersten n Elemente einer Liste zurück
newTake :: Int -> NewList a -> NewList a

-- Äquivalent zu (++) bei normalen Listen.
-- Fügt zwei NewLists aneinander (Infix)
-- z.B. funktioniert (newlistA ... newlistB)
-- genau wie (normallistA ++ normallistB)
(...) :: NewList a -> NewList a -> NewList a
```

Aufgabe 6-3 (ABGABE!)

a) Modellieren Sie mit Hilfe von `data` einen Typ `Auto`. Jedes Auto soll eine `bezeichnung (String)`, einen `preis (int)`, ein `alter (int)` und einen `besitzer (String)` haben (in dieser Reihenfolge, bitte Record Syntax verwenden). Der Besitzer soll dabei vom Typ `Person` sein:

```
data Person = Person {
                name :: String,
                adresse :: String
              }
```

b) Erweitern Sie den Datentypen `Person` so, dass eine Person in der Konsole ausgegeben werden kann und zwei Personen auf Gleichheit verglichen werden können.

c) Schreiben Sie folgende Funktionen:
   1. `printAuto` : Gibt einen formatierten String zurück, der das gegebene Auto beschreibt.
   2. `collectionWorth` : Erhält eine Person und eine Liste von Autos. Berechnet den Wert aller Autos aus der Liste, die der Person gehören.

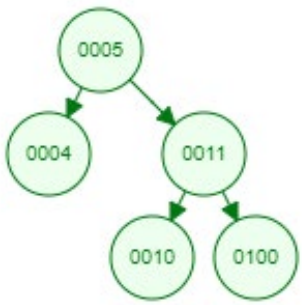
d) Schreiben Sie die Funktion `oldtimers`, die alle Autos aus einer gegebenen Liste zurückgibt, die mindestens 30 Jahre alt sind.

Aufg. 6.3 bitte bis 12.11.24, 23:59 Uhr in Moodle hochladen! Max. 2 Punkte!

Dateiname: Serie6.hs, als erste Zeile fügen Sie bitte ein: `module Serie6 where`

Aufgabe 6-4 (Data - Bäume)

a) Binäre Bäume sind Datenstrukturen, welche aus Knoten bestehen, genau einen Vorgänger (außer die Wurzel) und höchstens zwei Nachfolger haben. Erstellen Sie eine Datenstruktur: `data MyTree`, welche einen Binärbaum repräsentiert und definieren Sie eine Instanz, welche folgendermaßen aufgebaut ist:



b) Erstellen Sie eine Funktion `insertIntoTree :: (Ord a) => MyTree a -> a -> MyTree a`, welche ein Element in einen vorhandenen Binärbaum sortierbarer Elemente einfügt. Dabei soll sichergestellt

werden, dass keine Elemente eingefügt werden, die schon im Baum vorhanden sind und dass linke Kindknoten immer kleiner, rechte Kindknoten immer größer als der Elternknoten sind.

c) Erstellen Sie eine Funktion
`binarySearch :: (Ord a) => MyTree a -> a -> Bool`, die durch einen binären Suchbaum sortierbarer Elemente traversiert und entscheidet, ob ein Element im Baum vorhanden ist.