Programmierparadigmen (188350)

Vorleistung / Testat

Abgabefrist: Freitag, 27. Januar 2023.

Um zu der mündlichen Prüfung zugelassen zu werden, müssen Sie dieses Aufgabenblatt zufriedenstellend bearbeitet haben.

1 Umfang und Zweck

Sinn dieses Aufgabenblattes ist, sicherzustellen, dass Sie den Stoff hinreichend verstanden zu haben, um die mündliche Prüfung bestehen zu können. Auch soll es Ihnen ein Gefühl geben, um welche Art von Fragen es bei der Prüfung gehen wird (wobei mir klar ist, dass Sie für dieses Blatt zwei Wochen, in der Prüfung aber nur 20 Minuten haben).

Grundlegende Konzepte, die Sie verstanden haben müssen:

- Datentypen wie `int, string, char, bool` oder wie auch immer die in der jeweiligen Programmiersprache heißen.
- Zusammengesetzte Datentypen wie Tupel oder Listen, also `'(1 . "hallo")` und `'(3 4)` in Racket oder `(1 , "hallo")` und `[3, 4]` in Elm.
- Konzepte wie Funktionen, Parameter, Funktionsaufrufe, Argumente.
- Auswertung eines Ausdrucks (ob in Racket, λ-Kalkül oder Elm).

Erlaubte Hilfsmittel zur Bearbeitung dieses Blattes: Sie dürfen grundsätzlich im Internet nach technischen Fragen suchen, wie zum Beispiel "how to append two strings in racket?"

Nicht erlaubte Hilfsmittel: Sie dürfen keine Antworten von Anderen kopieren, abändern etc. In der Prüfung sind Sie ja auch alleine und können keine Kollegen fragen.

2 Programmieraufgaben

Exercise 1. Kennen Sie die Collatz-Vermutung? Definieren wir eine Funktion $c : \mathbb{N} \to \mathbb{N}$ via

$$c(n) := \begin{cases} n/2 & \text{falls } n \text{ gerade ist,} \\ 3n+1 & \text{falls } n \text{ ungerade ist.} \end{cases}$$

Für eine Zahl n können wir dann die Folge $n, c(n), c(c(n)), c(c(c(n))), ...$ betrachten, für $n = 3$ zum Beispiel

$$3, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, 4, 2, 1, \ldots$$

Sobald wir bei der 1 angelangt sind, wiederholt sich die Folge auf immer und ewig. Die bislang unbewiesene Collatz-Vermutung besagt, dass die obige Folge für alle n schlussendlich bei 1 angelangt.

Schreiben Sie eine Racket-Funktion, die die obige Folge als Liste ausgibt, bis wir zur Zahl 1 gelangen.

```
> (col 3)
'(3 10 5 16 8 4 2 1)
```

Machen Sie das gleiche in Elm (wenn Sie es bereits in Racket gelöst haben, sollte es in Elm kaum Mehraufwand sein). **Tip:** In Elm sollten Sie statt `n/2` die Integer-Division `n//2` verwenden, damit Elm "weiß", dass ein `Int` rauskommt.

Exercise 2. Schreiben Sie eine Racket-Funktion `secondSmallest`, die aus einer Liste von ganzen Zahlen das zweitkleinste Element herausholt. Beachten Sie, was geschieht, falls das kleinste Element mehrfach vorkommt.

```
> (secondSmallest '(3 2 1 4 4))
2
> (secondSmallest '(3 2 1 1 5))
2
```

Schreiben Sie die (fast) gleiche Funktion in Elm. Überlegen Sie, was Sie tun müssen, wenn es kein zweitkleinstes Element gibt, z.B. in `[2,2,2,2]`. Verwenden Sie den Datentyp `Maybe Int`.

```
> secondSmallest [3, 2, 1, 4, 4]
Just 1 : Maybe Int
> secondSamllest [2, 2, 2, 2]
Nothing : Maybe Int
```

Exercise 3. Schreiben Sie eine Funktion `merge`, die zwei (bereits aufsteigend sortierte) Listen von ganzen Zahlen zu einer einzigen, insgesamt sortieren Liste zusammenfügt. Sie dürfen sich aussuchen, ob Sie Racket oder Elm verwenden.

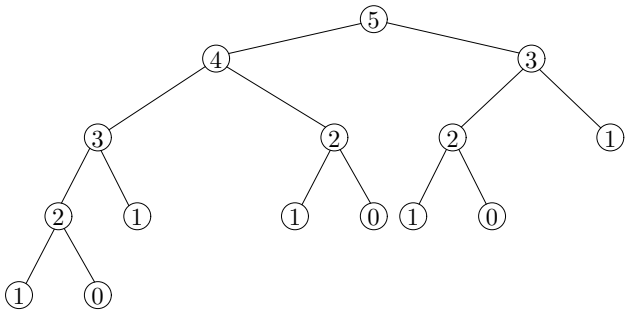
```
> merge [1,3,5,7] [2,4,4,5,6,8]
[1,2,3,4,4,5,5,6,7,8] : List Int
```

Exercise 4. Erinnern Sie sich an den Code, der die Fibonacci-Zahlen recht ineffizient berechnet:

```
fibRec : Int -> Int
fibRec n =
    if n <= 1 then
        n

    else
        fibRec (n - 1) + fibRec (n - 2)
```

Die Ausführung dieser Funktion erzeugt einen Rekursionsbaum, z.B. für `fibRec 5`:



Jeder Knoten steht für einen Aufruf der Funktion `fibRec`. Es ist ein *Binärbaum*: jeder Knoten hat entweder 0 oder 2 Kinder.

1. Wie würden Sie obigen Baum in Racket darstellen?
2. Entwerfen Sie einen Elm-Datenyp für Binärbäume. Erinnern Sie sich, wie man in Elm den Datentyp "verkettete Liste" bauen kann:

   ```
   type LinkedList
       = EmptyList
       | Link Int LinkedList
   ```

   Tun Sie das Entsprechende für Binärbäume: `type BinaryTree ...`
3. Schreiben Sie eine Elm-Funktion `buildFibRecursionTree : Int -> BinaryTree` die für eine gegebene Zahl n nicht die Fibonacci-Zahl F_n berechnet sondern den Rekursionsbaum baut, also für `buildFibRecursinoTree 5` z.B. den oben abgebildeten Baum.

3 Verständnisaufgaben

Exercise 5. In Racket können Sie beliebig verschachtelte Listen bauen:

```
> '(  1 (2 3) (4 (5) 5) (2 (3)))
```

Erklären Sie, warum das in Elm nicht geht, ja gar nicht gehen *darf*. Erkären Sie, wie man dennoch solche Strukturen bauen kann, wenn man sie braucht.

Exercise 6. Erklären Sie das Konzept des *Currying*. Eklären Sie es konkret am Beispiel

```
> List.map ( (++) "un" ) ["fertig", "schoen", "vollendet"]
```

Wenn dieser Code Sie verwirrt, fangen Sie an, herauszufinden, was `(++)` tut. Lesen Sie Kapitel 4.5, insbesondere das Code-Beispiel in Elm.