Aufgabe 1 Statische Semantik (__ /20 Punkte)

a) Füllen Sie die Lücken, sodass sich **gültige Aussagen der Statischen Semantik** ergeben. Alle vorkommenden Zahlen sind vom Typ `Nat`. Sie müssen **keine Beweisbäume** angeben!

Hinweis: In einigen Teilaufgaben gibt es mehrere richtige Lösungen. Die angegebene Lösung ist nur ein Beispiel.

i) $\emptyset \vdash$ `815 :` _____ **__ /10**

$\emptyset \vdash$ `815 : Nat`

ii) $\emptyset \vdash$ `if false then` _____ `else` _____ `: Bool`

$\emptyset \vdash$ `if false then true else false : Bool`

iii) $\emptyset \vdash$ `(4711, (if` _____ `then` _____ `else` _____ `)) : Nat * Nat`

$\emptyset \vdash$ `(4711, (if true then 4711 else 815)) : Nat * Nat`

iv) $\emptyset \vdash$ `let x = 1 in x <` _____ `:` _____

$\emptyset \vdash$ `let x = 1 in x < 4711 : Bool`

v) $\{$ `x` $\mapsto$ _____ $\}$ $\vdash$ `fun (y:` _____ `) -> x + y :` _____

$\{$ `x` $\mapsto$ `Nat` $\}$ $\vdash$ `fun (y: Nat) -> x + y : Nat` $\to$ `Nat`

b) Geben Sie einen **vollständigen Beweisbaum** für folgende **Aussage der Statischen Semantik** an: **__ /10**

$$\{\texttt{y} \mapsto \text{Nat}\} \vdash \texttt{fun (x: Nat)} \to \texttt{if x < 4711 then x else y} : \text{Nat} \to \text{Nat}$$

Definiere $\Sigma := \{\texttt{y} \mapsto \text{Nat}, \texttt{x} \mapsto \text{Nat}\}$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma \vdash \texttt{x} : \text{Nat}} \quad \dfrac{}{\Sigma \vdash \texttt{4711} : \text{Nat}}}{\Sigma \vdash \texttt{x < 4711} : \text{Bool}} \quad \dfrac{}{\Sigma \vdash \texttt{x} : \text{Nat}} \quad \dfrac{}{\Sigma \vdash \texttt{y} : \text{Nat}}}{\Sigma \vdash \texttt{if x < 4711 then x else y} : \text{Nat}}}{\{\texttt{y} \mapsto \text{Nat}\} \vdash \texttt{fun (x: Nat)} \to \texttt{if x < 4711 then x else y} : \text{Nat} \to \text{Nat}}$$

Aufgabe 2 Dynamische Semantik (__ /20 Punkte)

a) Füllen Sie die Lücken, sodass die Ausdrücke **typkorrekt** sind und sich **gültige Aussagen der Dynamischen Semantik** ergeben. Alle vorkommenden Zahlen sind vom Typ `Nat`. Sie müssen **keine Beweisbäume** angeben!

Hinweis: In einigen Teilaufgaben gibt es mehrere richtige Lösungen. Die angegebene Lösung ist nur ein Beispiel.

i) $\{\mathtt{p} \mapsto (4711, 815)\} \vdash \mathtt{if}$ ____ $\mathtt{then\ fst\ p\ else\ (fst\ p\ *\ snd\ p)} \Downarrow$ ____ __/10

$$\dfrac{\dfrac{}{\{\mathtt{p} \mapsto (4711,815)\} \vdash \mathtt{true} \Downarrow \mathtt{true}} \quad \dfrac{\dfrac{}{\{\mathtt{p} \mapsto (4711,815)\} \vdash \mathtt{p} \Downarrow (4711,815)}}{\{\mathtt{p} \mapsto (4711,815)\} \vdash \mathtt{fst\ p} \Downarrow 4711}}{\{\mathtt{p} \mapsto (4711,815)\} \vdash \mathtt{if\ \underline{true}\ then\ fst\ p\ else\ fst\ p\ *\ snd\ p} \Downarrow \underline{4711}}$$

$$\dfrac{\dfrac{}{\{\mathtt{p} \mapsto (4711,815)\} \vdash \mathtt{false} \Downarrow \mathtt{false}} \quad \dfrac{\dfrac{\dfrac{}{\{\mathtt{p} \mapsto (4711,815)\} \vdash \mathtt{p} \Downarrow (4711,815)}}{\{\mathtt{p} \mapsto (4711,815)\} \vdash \mathtt{fst\ p} \Downarrow 4711} \quad \dfrac{\dfrac{}{\{\mathtt{p} \mapsto (4711,815)\} \vdash \mathtt{p} \Downarrow (4711,815)}}{\{\mathtt{p} \mapsto (4711,815)\} \vdash \mathtt{snd\ p} \Downarrow 815}}{\{\mathtt{p} \mapsto (4711,815)\} \vdash \mathtt{fst\ p\ *\ snd\ p} \Downarrow 3839465}}{\{\mathtt{p} \mapsto (4711,815)\} \vdash \mathtt{if\ \underline{false}\ then\ fst\ p\ else\ fst\ p\ *\ snd\ p} \Downarrow \underline{3839465}}$$

ii) ____ $\vdash \mathtt{let\ f\ (x:\ Nat):\ Nat\ =\ x\ +\ a} \Downarrow$ ____

Damit der Ausdruck typkorrekt ist, muss der Bezeichner `a` an eine natürliche Zahl gebunden sein.

$$\dfrac{}{\underline{\{\mathtt{a} \mapsto 4711\}} \vdash \mathtt{let\ f\ (x:\ Nat):\ Nat\ =\ x\ +\ a} \Downarrow \underline{\{\mathtt{f} \mapsto \langle \{\mathtt{a} \mapsto 4711\}, \mathtt{x}, \mathtt{x\ +\ a}\rangle\}}}$$

iii) $\{\mathtt{x} \mapsto 4711, \mathtt{f} \mapsto \langle \{\mathtt{x} \mapsto 42\}, \mathtt{x}, \mathtt{x}\rangle\} \vdash \mathtt{x} \Downarrow$ ____

$$\dfrac{}{\{\mathtt{x} \mapsto 4711, \mathtt{f} \mapsto \langle \{\mathtt{x} \mapsto 42\}, \mathtt{x}, \mathtt{x}\rangle\} \vdash \mathtt{x} \Downarrow \underline{4711}}$$

iv) $\emptyset \vdash \mathtt{(fun\ x\ \to\ x\ 5)\ (fun\ y\ \to\ y\ +\ y)} \Downarrow$ ____

Definiere: $\delta_1 := \{\mathtt{x} \mapsto \langle \emptyset, \mathtt{y}, \mathtt{y\ +\ y}\rangle\}$ und $\delta_2 := \{\mathtt{y} \mapsto 5\}$

$$\dfrac{\dfrac{}{\emptyset \vdash \mathtt{fun\ x\ \to\ x\ 5} \Downarrow \langle \emptyset, \mathtt{x}, \mathtt{x\ 5}\rangle} \quad \dfrac{}{\emptyset \vdash \mathtt{fun\ y\ \to\ y\ +\ y} \Downarrow \langle \emptyset, \mathtt{y}, \mathtt{y\ +\ y}\rangle} \quad \dfrac{}{\delta_1 \vdash \mathtt{x} \Downarrow \langle \emptyset, \mathtt{y}, \mathtt{y\ +\ y}\rangle} \quad \dfrac{}{\delta_1 \vdash 5 \Downarrow 5} \quad \dfrac{\dfrac{}{\delta_2 \vdash \mathtt{y} \Downarrow 5} \quad \dfrac{}{\delta_2 \vdash \mathtt{y} \Downarrow 5}}{\delta_2 \vdash \mathtt{y\ +\ y} \Downarrow 10}}{\emptyset \vdash \mathtt{(fun\ x\ \to\ x\ 5)\ (fun\ y\ \to\ y\ +\ y)} \Downarrow \underline{10}}$$

v) $\{\mathtt{x} \mapsto 4711\} \vdash \mathtt{x\ >}$ ____ $\Downarrow \mathtt{true}$

$$\dfrac{\dfrac{}{\{\mathtt{x} \mapsto 4711\} \vdash \mathtt{x} \Downarrow 4711} \quad \dfrac{}{\{\mathtt{x} \mapsto 4711\} \vdash 42 \Downarrow 42}}{\{\mathtt{x} \mapsto 4711\} \vdash \mathtt{x\ >\ \underline{42}} \Downarrow \mathtt{true}}$$

b) **Wählen Sie** von den fünf Aussagen aus Teilaufgabe a) **eine aus** und geben Sie **nur für diese Aussage** einen **vollständigen Beweisbaum** mit den Regeln der Dynamischen Semantik an.
Tipp: Die Bäume für die fünf Aussagen werden unterschiedlich groß. Machen Sie sich kurz Gedanken darüber, welche Aussage zu einem kleinen Baum führt. __ /10
Sie können Ihren Beweisbaum oben in die Zwischenräume schreiben und dabei die eigentliche Aussage als Teil des Baumes verwenden. Falls Ihnen der Platz nicht ausreicht, können Sie die Rückseite benutzen.

Aufgabe 3 Entwurfsmuster (__ /20 Punkte)

Lösen Sie diese Aufgabe **funktional**, d. h. `mutable` und `ref` dürfen in Ihrer Lösung nicht vorkommen. Verwenden Sie **keine Bibliotheksfunktionen!**

a) Schreiben Sie eine Funktion `identity: Nat -> Nat`, die die Identitätsfunktion auf den natürlichen Zahlen implementiert, d.h. `identity n = n` für alle `n: Nat`. Gehen Sie **strikt nach Peano Entwurfsmuster** vor!

Beispiele:

```
identity 0 = 0        identity 1 = 1        identity 5 = 5        identity 4711 = 4711
```

__ /5

```
let rec identity (n : Nat) : Nat =
    if n = 0N then 0N
    else 1N + identity (n - 1N)
```

b) Schreiben Sie eine Funktion `evenOddSums: List<Nat> -> Nat * Nat`, die für eine Liste von natürlichen Zahlen die Summe der geraden und die Summe der ungeraden Zahlen in einem Tupel zurückgibt. Gehen Sie **strikt nach dem Listen-Entwurfsmuster** vor!

Beispiele:

```
evenOddSums [] = (0,0)                    evenOddSums [5;7;9] = (0,21)
evenOddSums [1;2;3;4] = (6,4)             evenOddSums [2;4;6] = (12,0)
```

__ /5

```
let rec evenOddSums (xs: List<Nat>): Nat * Nat =
    match xs with
    | [] -> (0N, 0N)
    | x :: xs' ->
        let (evenSum, oddSum) = evenOddSums xs'
        if x % 2N = 0N then
            (x + evenSum, oddSum)
        else
            (evenSum, x + oddSum)
```

c) Im Folgenden betrachten wir folgende Funktion:

```
let rec f (n: Nat): Nat = if n = 0 then 0 else f (f (n - 1))
```

Beantworten Sie die folgenden Fragen zu dieser Funktion. Geben Sie jeweils eine kurze Begründung an. **__ / 10**

1. Ist die Funktion nach dem Peano-Entwurfsmuster geschrieben?

   Nein. Erstens gibt es zwei (geschachtelte) rekursive Aufrufe. Zweitens wird im äußeren rekursiven Aufruf nicht `n - 1` verwendet.

2. Für welche Eingabewerte terminiert die Funktion? Was ist der Rückgabewert im Fall der Termination?

   Die Funktion terminiert für alle Eingabewerte `n` und gibt immer `0` zurück. Dies lässt sich durch vollständige Induktion über `n` zeigen: Für `n = 0` terminiert die Funktion offensichtlich mit Rückgabewert `0`. Für `n > 0` terminiert der innere Aufruf `f (n - 1)` nach Induktionsvoraussetzung mit Rückgabewert `0`. Somit ist der äußere Aufruf `f 0`, der entsprechend dem Basisfall der Funktion terminiert und `0` zurückgibt.

3. Angenommen, die Funktion terminiert für den Eingabewert `n`. Wie viele Rekursionsschritte werden verwendet, um den Rückgabewert zu berechnen?

   Die Anzahl der Rekursionsschritte ist `2n`. Dies lässt sich durch vollständige Induktion über `n` zeigen: Für `n = 0` sind keine Rekursionsschritte notwendig. Für `n > 0` terminiert der innere Aufruf `f (n - 1)` nach Induktionsvoraussetzung nach `2(n - 1)` Rekursionsschritten und gibt `0` zurück. Der äußere Aufruf `f 0` benötigt keinen weiteren Rekursionsschritt. Zusätzlich zu den `2(n - 1)` Rekursionsschritten des inneren Aufrufs werden also nur die zwei Aufrufe (innerer und äußerer) benötigt, was insgesamt `2n` Rekursionsschritte ergibt.

4. Könnte die Funktion so umgeschrieben werden, dass sie dem Peano-Entwurfsmuster entspricht und für alle Eingabewerte den gleichen Rückgabewert liefert wie die ursprüngliche Funktion? Wenn ja, geben Sie die umgeschriebene Funktion an. Wenn nein, begründen Sie warum nicht.

   Ja, die Funktion könnte wie folgt umgeschrieben werden:
   ```
   let rec f (n: Nat): Nat = if n = 0 then 0 else f (n - 1)
   ```

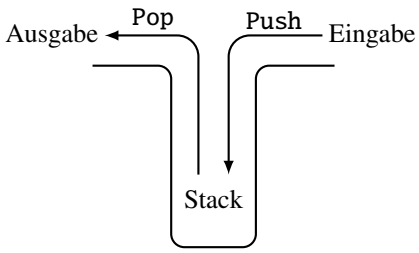
5. Was ändert sich, wenn die Funktion stattdessen wie folgt definiert wird?
   ```
   let rec f (n: Nat): Nat = if n = 0 then 1 else f (f (n - 1))
   ```

   Die Funktion terminiert nur für den Eingabewert `0`. Für den Eingabewert `1` haben wir den Zyklus `f 1` $\rightarrow$ `f (f 0)` $\rightarrow$ `f 1` $\rightarrow$ .... Für alle größeren Eingabewerte `n > 1` terminiert die Funktion ebenfalls nicht, da der innere Aufruf irgendwann den Wert `1` erreicht und somit in die gleiche Endlosschleife gerät.

Aufgabe 4 Stack-Sorting (__ / 20 Punkte)

Lösen Sie diese Aufgabe **funktional**, d. h. `mutable` und `ref` dürfen in Ihrer Lösung nicht vorkommen. Verwenden Sie **keine Bibliotheksfunktionen!**

Eine Liste heißt *stack-sortierbar*, wenn sie unter Verwendung einer sogenannten Stack-Maschine sortiert werden kann; siehe das schematische Diagramm rechts. Ein Stack-Maschinen-Programm ist durch eine Folge von `Push`- und `Pop`-Anweisungen gegeben, wobei `Push` ein Element vom Anfang der Eingabeliste auf den Stack legt und `Pop` das oberste Stack-Element an das Ende der Ausgabeliste hängt. Die Folge `[0; 1; 2]` ist zum Beispiel stack-sortierbar, wie das Programm `[Push; Pop; Push; Pop; Push; Pop]` zeigt – ein `Push` gefolgt von einem `Pop` verschiebt effektiv ein Element von der Eingabe zur Ausgabe.

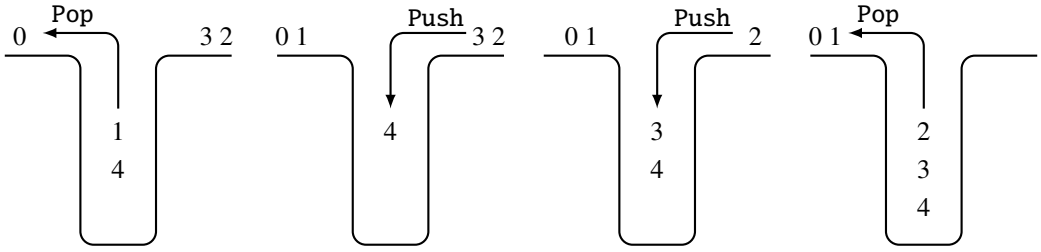


Die umgekehrte Folge `[2; 1; 0]` ist ebenfalls stack-sortierbar: `[Push; Push; Push; Pop; Pop; Pop]` sortiert die Eingabe – wir pushen zuerst alle Elemente und poppen sie dann. Nicht jede Permutation ist stack-sortierbar: `[1; 2; 0]` ist das kürzeste Gegenbeispiel.

Ein etwas größeres Beispiel ist die Liste `[4; 0; 1; 3; 2]`, die durch folgende Sequenz stack-sortiert wird:

`[Push; Push; Pop; Push; Pop; Push; Push; Pop; Pop; Pop]`.

Hier sind vier aufeinanderfolgende Momentaufnahmen des Sortiervorgangs, die jeweils den Zustand *vor* der Ausführung der Anweisung zeigen.



Schreiben Sie eine Funktion `assemble`, die zu einer gegebenen Liste von natürlichen Zahlen ein Stack-Programm konstruiert, das diese sortiert, sofern dies möglich ist. Wenn die Eingabe nicht stack-sortierbar ist, kann eine beliebige Anweisungsfolge zurückgegeben werden.

Beispiele:

```
assemble [0; 2; 1] = [Push; Pop; Push; Push; Pop; Pop]
assemble [2; 1; 0] = [Push; Push; Push; Pop; Pop; Pop]
assemble [4; 0; 1; 3; 2] = [Push; Push; Pop; Push; Pop; Push; Push; Pop; Pop; Pop]
```

```
let rec assemble (input: List<Nat>): List<Instruction> =
    let rec helper (input: List<Nat>) (stack: List<Nat>) : List<Instruction> =
        match input, stack with
        | [], [] -> []
        | [], (_ :: stack) -> Pop :: helper [] stack
        | (x :: xs), [] -> Push :: helper xs [x]
        | (x :: xs), (y :: ys) ->
            if x <= y then
                Push :: helper xs (x :: y :: ys)
            else
                Pop :: helper (x :: xs) ys
    helper input []
```

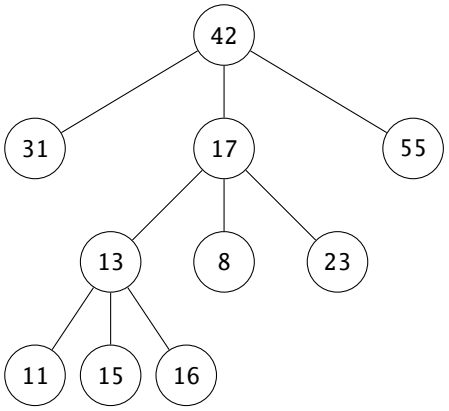
Aufgabe 5 Ternärbäume (__ /20 Punkte)

Sie dürfen zur Lösung dieser Aufgabe Bibliotheksfunktionen verwenden.

Gegeben ist der folgende Typ für ternäre Bäume:

```
type Tree<'a> = | Leaf | Node of 'a * Tree<'a> * Tree<'a> * Tree<'a>
```

Beispiel: Rechts sehen Sie den Code zu dem hier dargestellten Baum. Zur besseren Übersichtlichkeit sind die Blätter (`Leaf`s) nicht eingezeichnet.




```
let exTree: Tree<Nat> =
    Node(42,
        Node(31, Leaf, Leaf, Leaf),
        Node(17,
            Node(13,
                Node(11, Leaf, Leaf, Leaf),
                Node(15, Leaf, Leaf, Leaf),
                Node(16, Leaf, Leaf, Leaf)
            ),
            Node(8, Leaf, Leaf, Leaf),
            Node(23, Leaf, Leaf, Leaf)
        ),
        Node(55, Leaf, Leaf, Leaf)
    )
```

a) Schreiben Sie eine Funktion `size<'a>: Tree<'a> -> Nat`, die die Anzahl der Knoten eines Baums zurückgibt.

Beispiele:

```
size Leaf = 0
size (Node (2, Leaf, Node (1, Leaf, Leaf, Leaf), Leaf)) = 2
size exTree = 10
```

__/5

```
let rec size<'a> (t: Tree<'a>): Nat =
    match t with
    | Leaf -> 0N
    | Node (_, l, m, r) -> 1N + size l + size m + size r
```

b) Schreiben Sie eine Funktion `mirror<'a>: Tree<'a> -> Tree<'a>`, die einen Baum spiegelt.

Beispiele:

```
mirror Leaf = Leaf
mirror (Node (2, Leaf, Leaf, Node (1, Leaf, Leaf, Leaf)))
      = Node (2, Node (1, Leaf, Leaf, Leaf), Leaf, Leaf)
mirror exTree =
    Node(42,
        Node(55, Leaf, Leaf, Leaf),
        Node(17,
            Node(23, Leaf, Leaf, Leaf),
            Node(8, Leaf, Leaf, Leaf),
            Node(13,
                Node(16, Leaf, Leaf, Leaf),
                Node(15, Leaf, Leaf, Leaf),
                Node(11, Leaf, Leaf, Leaf))),
        Node(31, Leaf, Leaf, Leaf))
```

__/5

```
let rec mirror<'a> (t: Tree<'a>): Tree<'a> =
    match t with
    | Leaf -> Leaf
    | Node (x, l, m, r) -> Node (x, mirror r, mirror m, mirror l)
```

c) Schreiben Sie eine Funktion `partition: Nat -> List<Nat * Nat * Nat>`, die für eine gegebene Zahl `n` alle möglichen Tripel von Zahlen zurückgibt, die in Summe `n` ergeben.

Hinweis: Duplikate in der Ausgabe sind erlaubt.

Beispiele (Reihenfolge ist jeweils egal und Duplikate sind erlaubt):

```
partition 0 = [(0, 0, 0)]
partition 1 = [(0, 0, 1); (0, 1, 0); (1, 0, 0)]
partition 2 =
    [(0, 0, 2); (0, 1, 1); (0, 2, 0); (1, 0, 1); (1, 1, 0); (2, 0, 0)]
```

__/5

```
let rec partition (n: Nat): List<Nat*Nat*Nat> =
    if n = 0N then [(0N,0N,0N)]
    else partition (n - 1N)
         |> List.collect (fun (x,y,z) -> [(x+1N,y,z); (x,y+1N,z); (x,y,z+1N)])
```

d) Schreiben Sie eine Funktion `allOfSize: Nat -> List<Tree<Unit>>`, die alle ternären Bäume der gegebenen Größe (Anzahl der Knoten) zurückgibt.

Hinweis: Verwenden Sie die Funktion `partition`. Es ist wieder erlaubt, Duplikate in der Ausgabe zu haben. Es kann hilfreich sein, Listenbeschreibungen zu verwenden.

Beispiele:

```
allOfSize 0 = [Leaf]
allOfSize 1 = [Node ((), Leaf, Leaf, Leaf)]
allOfSize 2 =
    [Node ((), Leaf, Leaf, Node ((), Leaf, Leaf, Leaf));
     Node ((), Leaf, Node ((), Leaf, Leaf, Leaf), Leaf);
     Node ((), Node ((), Leaf, Leaf, Leaf), Leaf, Leaf)]
```

__/5

```
let rec allOfSize (n: Nat): List<Tree<Unit>> =
    if n = 0N then [Leaf]
    else [ for (x, y, z) in partition (n - 1N) do
           for l in allOfSize x do
           for m in allOfSize y do
           for r in allOfSize z do
               Node ((), l, m, r) ]
```

Aufgabe 6 Reguläre Ausdrücke (__ /20 Punkte)

a) Wir betrachten den regulären Ausdruck

$$(a \cdot b)^* \cdot b \cdot \Big((a \cdot a \cdot b) \mid (b \cdot a \cdot a)\Big)^* \cdot a$$

über dem Alphabet $\{a, b\}$.

Kreuzen Sie an, ob die folgenden Wörter in der von dem Ausdruck beschriebenen Sprache enthalten sind oder nicht. Für richtige Antworten erhalten Sie einen Punkt, für falsche Antworten wird ein Punkt abgezogen. Nicht markierte Zeilen wirken sich nicht auf die Punktzahl aus. Diese Teilaufgabe wird mit mindestens 0 Punkten bewertet.

__ /6

Wort	**enthalten**	**nicht enthalten**
abababababa		X
abbaabbaabbaa		X
baabbaabaaaab		X
bbbbbbbbbbbba		X
abbbaaaabaaba	X	
abababbbbaaa	X	

b) Die durch einen regulären Ausdruck beschriebene Sprache lässt sich umgangssprachlich beschreiben. Beispiele über dem Alphabet $\{a, b\}$ sind:

$((a \cdot b) \mid (a \cdot a) \mid (b \cdot b) \mid (b \cdot a))^*$	Die Sprache aller Wörter gerader Länge.
$(a \mid b)^* \cdot b \cdot (a \mid b)$	Die Sprache aller Wörter, die an der vorletzten Position ein b enthalten.
$(b \mid (a \cdot b^* \cdot a))^*$	Die Sprache aller Wörter mit einer geraden Anzahl von as.

Geben Sie eine umgangssprachliche Beschreibung der folgenden regulären Ausdrücke an. Beschreiben Sie die durch die regulären Ausdrücke beschriebene Sprache jeweils so knapp und präzise wie möglich ohne dabei zu paraphrasieren.

Alphabet: $\{a, b, c\}$

__/6

1. $(a \mid b)^* \cdot c \cdot (a \mid b)^*$

   Die Sprache aller Wörter, die genau ein c enthalten.

2. $a \cdot (a \cdot b)^* \cdot (a \mid b \mid c)^* \cdot (b \cdot c)^* \cdot c$

   Die Sprache aller Wörter, die mit einem a beginnen und einem c enden.

3. $((a \cdot a) \mid (b \cdot b))^* \cdot \Big(((a \cdot b) \mid (b \cdot a)) \cdot ((a \cdot a) \mid (b \cdot b))^* \cdot ((a \cdot b) \mid (b \cdot a)) \cdot ((a \cdot a) \mid (b \cdot b))^*\Big)^*$

   Die Sprache aller Wörter, die eine gerade Anzahl as und eine gerade Anzahl bs enthalten.

c) Bestimmen Sie die folgenden Rechtsfaktoren. Geben Sie in der Rechnung **jeweils den ersten Schritt explizit** an, nachfolgende Zwischenschritte dürfen Sie zusammenfassen.

Alphabet: $\{a, b, c\}$

__/8

$$
\begin{aligned}
a \setminus \left((a \cdot a) \mid (b \cdot b))^*\right) &= (a \setminus (a \cdot a \mid b \cdot b)) \cdot (a \cdot a \mid b \cdot b)^* \\
&= ((a \setminus a \cdot a) \mid (a \setminus b \cdot b)) \cdot (a \cdot a \mid b \cdot b)^* \\
&= ((a \setminus a) \cdot a \mid (a \setminus b) \cdot b) \cdot (a \cdot a \mid b \cdot b)^* \\
&= (\epsilon \cdot a \mid \emptyset \cdot b) \cdot (a \cdot a \mid b \cdot b)^* \\
&= (a \mid \emptyset) \cdot (a \cdot a \mid b \cdot b)^* \\
&= a \cdot (a \cdot a \mid b \cdot b)^*
\end{aligned}
$$

$$
\begin{aligned}
b \setminus \left((b \cdot c)^* \cdot b\right) &= (b \setminus (b \cdot c)^*) \cdot b \mid (b \setminus b) \\
&= (b \setminus b \cdot c) \cdot (b \cdot c)^* \cdot b \mid \epsilon \\
&= (b \setminus b) \cdot c \cdot (b \cdot c)^* \cdot b \mid \epsilon \\
&= \epsilon \cdot c \cdot (b \cdot c)^* \cdot b \mid \epsilon \\
&= c \cdot (b \cdot c)^* \cdot b \mid \epsilon
\end{aligned}
$$

$$
\begin{aligned}
c \setminus \left((a \cdot b^*) \mid (b \cdot c^*)\right) &= (c \setminus a \cdot b^*) \mid (c \setminus b \cdot c^*) \\
&= (c \setminus a) \cdot b^* \mid (c \setminus b) \cdot c^* \\
&= \emptyset \cdot b^* \mid \emptyset \cdot c^* \\
&= \emptyset \mid \emptyset \\
&= \emptyset
\end{aligned}
$$

$$
\begin{aligned}
b \setminus \left((a \cdot b^* \cdot a) \mid b)^*\right) &= (b \setminus (a \cdot b^* \cdot a \mid b)) \cdot (a \cdot b^* \cdot a \mid b)^* \\
&= ((b \setminus a \cdot b^* \cdot a) \mid (b \setminus b)) \cdot (a \cdot b^* \cdot a \mid b)^* \\
&= ((b \setminus a) \cdot b^* \cdot a \mid \epsilon) \cdot (a \cdot b^* \cdot a \mid b)^* \\
&= (\emptyset \cdot b^* \cdot a \mid \epsilon) \cdot (a \cdot b^* \cdot a \mid b)^* \\
&= (\emptyset \mid \epsilon) \cdot (a \cdot b^* \cdot a \mid b)^* \\
&= \epsilon \cdot (a \cdot b \cdot a \mid b)^* \\
&= (a \cdot b^* \cdot a \mid b)^*
\end{aligned}
$$

Aufgabe 7 Imperative Programmierung mit Funktionalen (__ /20 Punkte)

In der Vorlesung wurde das Konzept der *Abstraktion über die rekursiven Aufrufe* einer Funktion eingeführt. Daraus resultiert ein sog. *Funktional* (eine *Funktion höherer Ordnung*). Das Funktional für die Fibbonacci Funktion (zur Erinnerung: `let rec fib (n : Nat) : Nat = if n <= 1 then n else fib (n - 1) + fib (n - 2)`) ist folgendes:

```
let FIB : (Nat -> Nat) -> (Nat -> Nat) =
   fun fib -> fun n -> if n <= 1 then n else fib (n - 1) + fib (n - 2)
```

a) Gegeben ist folgende Funktion:

```
let rec add (m : Nat) (n : Nat) : Nat =
    if n = 0 then m else 1 + add m (n - 1)
```

Geben Sie den Typ und den Rumpf des Funktionals für `add` an.

__ /2

```
let ADD : (Nat -> Nat -> Nat) -> (Nat -> Nat -> Nat) =
    fun add -> fun n m -> if n = 0 then m else 1 + add m (n - 1)
```

b) Schreiben Sie eine Funktion `fix<'a,'b> : (('a -> 'b) -> 'a -> 'b) -> ('a -> 'b)`, die ein Funktional `F` in die entsprechende rekursive Funktion `f` überführt. Es soll gelten: `fix F = f` (beispielsweise `fix FIB = fib`).

__ /6

```
let rec fix<'a, 'b> (functional : ('a -> 'b) -> ('a -> 'b)) (x : 'a) : 'b =
    functional (fix functional) x
    // Alternativlösung: let rec f (a : 'a) : 'b = functional f a in f x
```

c) Schreiben Sie eine Funktion `recCount<'a,'b> : (('a -> 'b) -> 'a -> 'b) -> ('a -> 'b * Nat)`, die ein Funktional in eine Funktion umwandelt, die das Funktional iteriert, und am Ende die Anzahl der bei der Auswertung getätigten rekursiven Aufrufe zusammen mit dem Ergebnis zurückgibt.

__/12

Beispiele:

`recCount FIB 2 = (1, 3)` `recCount FIB 3 = (2, 5)`

```
let recCount<'a, 'b> (fctnl : ('a -> 'b) -> ('a -> 'b)) : 'a -> 'b * Nat =
  let counter = ref 0
  let rec countF (a : 'a) : 'b =
      counter := !counter + 1;
      fctnl countF a
  in
    fun a -> (countF a , !counter)
```

Aufgabe 8 Ausnahmen (__ /20 Punkte)

a) Unter Berücksichtigung dieser Typ- und Ausnahmedefinitionen

```
type BA = | A of Bool | B

exception G of Nat
exception H of Bool
```

betrachten wir den folgenden Ausdruck. Dabei ist `f` eine Funktion vom Typ `Unit -> BA`.

```
try
    match f() with
    | B   -> 4711
    | A x -> if x then raise (G 2) else raise (H x)
with
| G x -> if x > 1 then x else raise (H true)
| H x -> if not x then raise (G 12) else 815
```

Bestimmen Sie für die folgenden fünf Implementierungen der Funktion `f` jeweils, zu welchem Wert obiger Ausdruck auswertet. Notieren Sie geworfene Ausnahmen dabei, wie in der Vorlesung eingeführt, mit einem Kästchen, die durch `raise (G 123)` geworfene Ausnahme also durch `G 123`.

1. `let f() = B` **__ /10**

   `4711`

2. `let f() = A true`

   `2`

3. `let f() = A false`

   `G 12`

4. `let f() = raise (G 1)`

   `H true`

5. `let f() = raise (H false)`

   `G 12`

b) Der Wert `false : Bool` ist das sogenannte *Nullelement* der Konjunktion &&. Für alle `b` gilt:

```
b && false = false = false && b.
```

Ist ein einziger Wert einer Liste, die wir mit && „aufsammeln“ wollen, **`false`**, so wissen wir bereits, dass das Endergebnis **`false`** ist, ohne dass wir den Rest der Liste betrachten müssen.

Schreiben Sie eine Funktion `all : List<Bool> -> Bool`, die prüft, ob alle Elemente der Liste **`true`** sind und die sofort mit **`false`** terminiert, sobald ein Wert in der Liste **`false`** ist. Verwenden Sie hierzu **`try ... with`** und Ausnahmen.

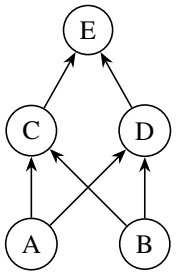
Hinweis: Definieren Sie eine (lokale) Hilfsfunktion.

__/10

```
exception Zero

let all (xs : List<Bool>) : Bool =
    let rec all' (xs : List<Bool>) : Bool =
        match xs with
        | [] -> true
        | x :: xs -> if x then all' xs else raise Zero
    in try all' xs with | Zero -> false
```

Aufgabe 9 Untertypen (__ /20 Punkte)



Es gelten die Untertypbeziehungen A $\preccurlyeq$ C, A $\preccurlyeq$ D, B $\preccurlyeq$ C, B $\preccurlyeq$ D, C $\preccurlyeq$ E und D $\preccurlyeq$ E, die in der nebenstehenden Abbildung visualisiert sind.

In der folgenden Tabelle werden je zwei Typen in Relation gesetzt:

- $t_1 \prec t_2$ bedeutet, dass t_1 ein Untertyp von t_2 ist ($t_1 \preccurlyeq t_2$), nicht jedoch t_2 ein Untertyp von t_1.
- $t_1 \succ t_2$ bedeutet, dass t_2 ein Untertyp von t_1 ist ($t_2 \preccurlyeq t_1$), nicht jedoch t_1 ein Untertyp von t_2.
- $t_1 \parallel t_2$ bedeutet, dass t_1 und t_2 unvergleichbar sind, das heißt, es gilt weder $t_1 \preccurlyeq t_2$ noch $t_2 \preccurlyeq t_1$.

Füllen Sie die Lücken in der Tabelle aus. In der ersten und dritten Spalte müssen Sie **einen** Typ eintragen, in der zweiten Spalte eine Relation ($\prec$ oder $\succ$ oder $\parallel$).

Für richtige Antworten (ganze Lücke) erhalten Sie zwei Punkte, für falsche Antworten werden zwei Punkte abgezogen. Nicht ausgefüllte Zeilen wirken sich nicht auf die Punktzahl aus. Diese Aufgabe wird mit mindestens 0 Punkten bewertet.

Zur Erinnerung: Die folgenden Deduktionsregeln gelten für die Relation »$\preccurlyeq$«:

$$\frac{t_1 \preccurlyeq t_2 \qquad t_2 \preccurlyeq t_3}{t_1 \preccurlyeq t_3} \qquad \frac{t_1 \preccurlyeq t_1' \qquad t_2 \preccurlyeq t_2'}{t_1 * t_2 \preccurlyeq t_1' * t_2'} \qquad \frac{t_1' \preccurlyeq t_1 \qquad t_2 \preccurlyeq t_2'}{t_1 \to t_2 \preccurlyeq t_1' \to t_2'}$$

t_1	Relation	t_2
C	$\succ$	A
C $*$ A	$\prec$	C $*$ C, C $*$ D, C $*$ E, E $*$ A, E $*$ C, E $*$ D, E $*$ E
A $*$ E	$\parallel$	B $*$ D
C $\to$ C	$\succ$	E $\to$ C, C $\to$ A, C $\to$ B, E $\to$ B, E $\to$ A
(D $*$ B) $\to$ E	$\prec$	(B $*$ B) $\to$ E, (A $*$ B) $\to$ E
A $\to$ C	$\parallel$	D $\to$ E
C $\to$ D	$\prec$	A $\to$ E
E $\to$ A	$\parallel$	Es gibt unendlich viele Lösungen, z.B. A. Falsch sind alle Typen der Form [ ] $\to$ [ ], in denen in der zweiten Box nicht B steht.
D $\to$ B $\to$ E	$\prec$	B $\to$ B $\to$ E, A $\to$ B $\to$ E
(A $\to$ D) $\to$ B	$\succ$	(A $\to$ E) $\to$ B

Aufgabe 10 Akzeptoren, Objektorientiert (__ /20 Punkte)

Aus der Vorlesung sind Akzeptoren für Sprachen bekannt. Wir modellieren Akzeptoren über einem Alphabet `'A` objektorientiert wie folgt:

```
type Recognizer<'A> =
  interface
    abstract member Accepting  : Bool
    abstract member Read : 'A -> Unit
  end
```

Die Eigenschaft `Accepting` gibt zurück, ob das leere Wort in der Sprache ist. Die Methode `Read` liest einen Buchstaben b des Alphabets ein und überführt einen Akzeptor `a` für Sprache L über in einen für $b \setminus L$.

a) Schreiben Sie einen Konstruktor `even0s : unit -> Recognizer<Bool>` für einen Akzeptor für die Sprache der Wörter über dem Alphabet `Bool`, die eine gerade Anzahl an **`false`**'s haben.

__ /5

```
let even0s () : Recognizer<Bool> =
    let mutable even0s  = true
    {  new Recognizer<Bool> with
       member _.Accepting = even0s
       member _.Read c = if c = false then even0s <- not even0s
    }
```

b) Schreiben Sie eine Funktion `decide (a : Recognizer<'A>) (w : List<'A>) : Bool`, die zurückgibt, ob das Wort `w` in der Sprache von `a`, ist. Zum Beispiel soll gelten:

__ /5

```
decide (even0s ()) [true ; false] = false
decide (even0s ()) [] = true
decide (even0s ()) [true ; false ; true ; false] = true
```

```
let rec decide (a : Recognizer<'A>) (w : List<'A>) : Bool =
  match w with
  | [] -> a.Accepting
  | x :: xs -> a.Read x ; decide a xs
```

c) Schreiben Sie eine Funktion `negate: (a : Recognizer<'A>) -> Recognizer<'A>`, die einen Akzeptor für die Sprache der Wörter, die *nicht* von `a` akzeptiert werden, zurückgibt.

__/2

```
let negate (a : Recognizer<'A>) : Recognizer<'A> =
   { new Recognizer<'A> with
     member _.Accepting = not a.Accepting
     member _.Read c = a.Read c
   }
```

d) Schreiben Sie eine Funktion `both: (a: Recognizer<'A>) -> (b: Recognizer<'A>) -> Recognizer<'A>`, die einen Akzeptor für die Sprache der Wörter, die von sowohl `a` als auch `b` akzeptiert werden, zurückgibt.

__/2

```
let both (a : Recognizer<'A>) (b : Recognizer<'A>) : Recognizer<'A> =
   { new Recognizer<'A> with
     member _.Accepting = a.Accepting && b.Accepting
     member _.Read c = a.Read c ; b.Read c
   }
```

e) Schreiben Sie eine Funktion `interleave: (a: Recognizer<'A>) -> (b: Recognizer<'A>) -> Recognizer<'A>`; Seien L_a die Sprache von `a` und L_b die Sprache von `b`. Die Funktion soll einen Akzeptor für die Sprache L_1 von Wörtern der Sprache L_a „verflochten“ mit Wörtern der Sprache L_b zurückgeben. Formal:

$$L_1 := \{a_1 b_1 a_2 b_2 \dots a_n b_n \mid n \in \mathbb{N}, a_i, b_i \in {}'\mathtt{A} \wedge a_1 \dots a_n \in L_a \wedge b_1 \dots b_n \in L_b\}$$

__/6

```
let interleave (a : Recognizer<'A>) (b : Recognizer<'A>) : Recognizer<'A> =
    let mutable asTurn = true
    { new Recognizer<'A> with
      member _.Accepting = asTurn && a.Accepting && b.Accepting
      member _.Read c = do
        if asTurn then a.Read c else b.Read c
        asTurn <- not asTurn
    }
```