Aufgabe 1 Statische Semantik (__ /20 Punkte)

Bei allen Zahlen, die in den Ausdrücken in dieser Aufgabe vorkommen, handelt es sich um Konstanten vom Typ `Nat`.

a) Geben Sie jeweils an, welchen Typ die folgenden Ausdrücke haben (**ohne Beweisbaum**).

1. `let x = 3 in let y = x + 5 in y % x` __ /6

   `Nat`

2. `let f (x: Nat) = x > 7 in f 3`

   `Bool`

3. `(fun (x: Nat) -> (fun (y: Nat) -> x + y)) 2`

   `Nat -> Nat`

b) Setzen Sie in die Lücken jeweils einen Teilausdruck ein, sodass sich insgesamt ein typkorrekter Ausdruck ergibt.

1. `fun (a: Bool) -> if a then 4711 else` _____ __ /7

   Teilausdruck vom Typ `Nat`, z.B. `815`.

2. `match` _____ `with | [] -> true | x -> false`

   Liste eines beliebigen Typs, z.B. `[]`, `[2]`, `[2;4;6]`, `[false]`.

3. `let x =` _____ `in 5 + fst x + snd x`

   Teilausdruck vom Typ `Nat * Nat`, z.B. `(1, 2)`.

4. `type Rectangle = {height: Nat; width: Nat; filled: Bool}`

   `let r =` _____ `in r.filled`

   Teilausdruck vom Typ `Rectangle`, z.B. `{height=1; width=2; filled=true}`.

c) Bestimmen Sie den Typ des Ausdrucks __ /7

$$\texttt{x + (f true)}$$

bezüglich der Signatur

$$\Sigma = \{\texttt{f} \mapsto \texttt{Bool} \to \texttt{Nat}, \texttt{x} \mapsto \texttt{Nat}\}$$

Geben Sie einen **vollständigen Beweisbaum** auf Grundlage der Regeln der **statischen Semantik** aus der Vorlesung an.

$$\dfrac{\dfrac{}{\Sigma \vdash \texttt{x} : \texttt{Nat}} \qquad \dfrac{\dfrac{}{\Sigma \vdash \texttt{f} : \texttt{Bool} \to \texttt{Nat}} \qquad \dfrac{}{\Sigma \vdash \texttt{true} : \texttt{Bool}}}{\Sigma \vdash \texttt{f true} : \texttt{Nat}}}{\Sigma \vdash \texttt{x + (f true)} : \texttt{Nat}}$$

Aufgabe 2 Dynamische Semantik (__ /20 Punkte)

a) Geben Sie jeweils an, zu welchem Wert die folgenden Ausdrücke auswerten (ohne Rechnung, Endergebnis genügt):

__ /6

1. `(if 99 % 2 = 0 then 7 else 1) + 2`

   3

2. `let f (x: Nat): Nat = if x < 4 then x + 3 else x * x in f 4`

   16

3. `(fun (f: Nat -> Nat -> Nat) -> f 815 4711) (fun (x: Nat) -> (fun (y: Nat) -> x))`

   815

b) Setzen Sie in die Lücken jeweils einen typkorrekten Teilausdruck ein, sodass der Gesamtausdruck zum angegebenen Ergebnis auswertet.

__ /4

1. $\emptyset \vdash$ `let x = (3 > 1, ____) in snd x && fst x` $\Downarrow$ `true`

   `true`

2. $\{\mathsf{g} \mapsto \langle \{z \mapsto 7\}, x, x + z \rangle\} \vdash \mathsf{g}$ ____ $\Downarrow 12$

   5

c) Werten Sie den Ausdruck `f (if a then 7 else 5)` bezüglich der Umgebung

__ /10

$$\delta := \{\mathsf{a} \mapsto \mathit{false}, \mathsf{f} \mapsto \langle \{\mathsf{a} \mapsto \mathit{false}\}, \mathtt{x}, \mathtt{x * 3} \rangle\}$$

aus. Geben Sie einen vollständigen Beweisbaum auf Grundlage der Auswertungsregeln aus der Vorlesung an.

Tipp: Legen Sie das Blatt quer und zeigen Sie $\delta \vdash$ `f (if a then 7 else 5)` $\Downarrow \ldots$

definiere aus Platzgründen $\delta_2 := \{\mathsf{a} \mapsto \mathit{false}, \mathtt{x} \mapsto 5\}$

$$\dfrac{\dfrac{}{\delta \vdash \mathtt{f} \Downarrow \langle \{\mathsf{a} \mapsto \mathit{false}\}, \mathtt{x}, \mathtt{x * 3} \rangle} \quad \dfrac{\dfrac{}{\delta \vdash \mathtt{a} \Downarrow \mathit{false}} \quad \dfrac{}{\delta \vdash 5 \Downarrow 5}}{\delta \vdash \mathtt{if\ a\ then\ 7\ else\ 5} \Downarrow 5} \quad \dfrac{\dfrac{}{\delta_2 \vdash \mathtt{x} \Downarrow 5} \quad \dfrac{}{\delta_2 \vdash 3 \Downarrow 3}}{\delta_2 \vdash \mathtt{x * 3} \Downarrow 15}}{\delta \vdash \mathtt{f\ (if\ a\ then\ 7\ else\ 5)} \Downarrow 15}$$

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 die Funktion `interval: Nat -> List<Nat>`, welche für eine gegebene natürliche Zahl `n` die Liste `[n; ...; 0]` berechnet. Gehen Sie **strikt nach Peano Entwurfsmuster** vor.

Beispiele:

```
interval 0 = [0]          interval 1 = [1; 0]          interval 2 = [2; 1; 0]
```

__ /5

```
let rec interval (n: Nat): List<Nat> =
    if n = 0N then [0N]
    else n :: interval (n - 1N)
```

b) Schreiben Sie die Funktion `ntimes<'a>: ('a -> 'a) -> 'a -> Nat -> 'a`, welche eine Funktion `f` vom Typ `'a -> 'a`, einen Startwert `x` vom Typ `'a` sowie eine natürliche Zahl `n` nimmt und die Funktion `f` auf `x` `n`-mal anwendet. Für `n = 0` soll `x` unverändert zurückgegeben werden. Gehen Sie **strikt nach Peano Entwurfsmuster** vor.

Beispiele:

```
ntimes (fun x -> x * x) 3 0 = 3          ntimes (fun x -> x * 2) 1 5 = 32
```

__ /5

```
let rec ntimes<'a> (f: 'a -> 'a) (x: 'a) (n: Nat): 'a =
    if n = 0N then x
    else f (ntimes f x (n - 1N))
```

Für die nächsten beiden Teilaufgaben verwenden wir folgenden Typen für Binärbäume mit Daten in den Blättern:

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

c) Schreiben Sie die Funktion `completeOfHeight: Nat -> Tree<Unit>`, die für eine gegebene natürliche Zahl `n` den Baum der Höhe `n` berechnet, der die maximal mögliche Anzahl an Blättern besitzt. Gehen Sie **strikt nach Peano Entwurfsmuster** vor. Sie dürfen **nur einen rekursiven Aufruf** verwenden.

Beispiele:

```
completeOfHeight 0 = Leaf ()
completeOfHeight 1 = Node (Leaf (), Leaf ())
completeOfHeight 2 = Node (Node (Leaf (), Leaf ()), Node (Leaf (), Leaf ()))
```

__/5

```
let rec completeOfHeight (n: Nat): Tree<Unit> =
    if n = 0N then Leaf ()
    else
        let t = completeOfHeight (n - 1N)
        Node (t, t)
```

d) Schreiben Sie die Funktion `numberOfLeaves<'a>: Tree<'a> -> Nat`, welche die Anzahl der Blätter im gegebenen Baum bestimmt. Gehen Sie **nach Struktur Entwurfsmuster des Datentyps** vor.

Beispiele:

```
numberOfLeaves (Leaf 5) = 1
numberOfLeaves Node (Leaf 5, Leaf 3) = 2
numberOfLeaves Node (Leaf 7, Node (Leaf 5, Leaf 3)) = 3
```

__/5

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

Aufgabe 4 ConsSnoc Folgen (__ /20 Punkte)

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

Wir betrachten den folgenden Datentyp, um Folgen von Elementen darzustellen:

```
type Sequ<'T> =
    | Nil
    | Cons of 'T * Sequ<'T>
    | Snoc of Sequ<'T> * 'T
```

Der Vorteil dieses Datentyps gegenüber herkömmlichen Listen liegt darin, dass man in konstanter Zeit Elemente an beiden Enden der Folge hinzufügen kann:

- `Nil` ist die leere Folge,
- `Cons (x, xs)` ist die Folge, die `x` gefolgt von den Elementen aus der Folge `xs` enthält,
- `Snoc (xs, x)` ist die Folge, die die Elemente aus der Folge `xs` gefolgt von `x` enthält.

Somit ist `Snoc (Cons (1, Cons (2, Nil)), 3)` die Folge mit den Zahlen `1`, `2` und `3`.

Hinweis: Muster mit dem Listenkonstruktor `::` sowie die Funktion `@` passen *nicht* auf den hier definierten Datentyp!

a) Schreiben Sie eine Funktion `sum`, die eine Folge nimmt und deren Elemente summiert. Die Summe der leeren Folge sei als `0` definiert.

Beispiel: `sum (Snoc (Cons (9, Cons (8, Nil)), 7)) = 24`

__ /6

```
let rec sum (xs: Sequ<Nat>): Nat =
    match xs with
    | Nil -> 0N
    | Cons (x, xs') | Snoc (xs', x) -> x + sum xs'
```

Beachten Sie bei den folgenden beiden Teilaufgaben, dass die ConsSnoc Repräsentation nicht eindeutig ist. Zur Repräsentation der Liste `[1; 2]` als Folge vom Typ `Sequ<Nat>` gibt es vier verschiedene Möglichkeiten:

- `Cons (1, Cons (2, Nil))`
- `Cons (1, Snoc (Nil, 2))`
- `Snoc (Cons (1, Nil), 2)`
- `Snoc (Snoc (Nil, 1), 2)`

Für die von Ihrer Funktion zurückgegebene Folge können Sie selbst entscheiden, welche Repräsentation Sie benutzen wollen. Es gibt mehrere mögliche Lösungen.

b) Schreiben Sie eine Funktion `tail`, die eine Folge `xs` nimmt und eine Folge zurückgibt, die die Elemente aus `xs` ohne das erste Element enthält. Die Reihenfolge der Elemente in der zurückgegebenen Folge soll der Reihenfolge aus `xs` entsprechen. Wenn die Folge `xs` leer ist, dann soll `None` zurückgegeben werden.

Tipp: Gehen Sie streng nach dem Entwurfsmuster für den Typ `Sequ<'T>` vor.

Beispiele:

```
tail Nil = None
tail (Cons (7, Snoc (Nil, 2))) = Some (Snoc (Nil, 2))
tail (Snoc (Nil, 7)) = Some Nil
tail (Snoc (Cons (42, Cons (13, Nil)), 7)) = Some (Snoc (Cons (13, Nil), 7))
```

__/7

```
let rec tail<'a> (xs: Sequ<'a>): Sequ<'a> option =
    match xs with
    | Nil -> None
    | Cons (_, xs') -> Some xs'
    | Snoc (xs', x) ->
        match tail xs' with
        | Some z -> Some (Snoc (z, x))
        | None -> Some Nil
```

c) Schreiben Sie eine Funktion append, die zwei Folgen nimmt und deren Konkatenation berechnet, also eine Folge, die erst die Elemente aus der ersten und dann die Elemente aus der zweiten gegebenen Folge enthält.

__/7

```
// Variante 1: Rekursion über xs
// Hierbei wird xs zu einer reinen Cons Folge umgeformt und schließlich ys
// statt Nil unverändert als Restfolge eingesetzt.
let rec append<'a> (xs: Sequ<'a>) (ys: Sequ<'a>): Sequ<'a> =
    match xs with
    | Nil -> ys
    | Cons (x, xs') -> Cons (x, append xs' ys)
    | Snoc (xs', x) -> append xs' (Cons (x, ys))

// Variante 2: Rekursion über ys (symmetrisch zu Variante 1)
// Hierbei wird ys zu einer reinen Snoc Folge umgeformt und schließlich xs
// statt Nil unverändert als Anfangsfolge eingesetzt.
let rec append'<'a> (xs: Sequ<'a>) (ys: Sequ<'a>): Sequ<'a> =
    match ys with
    | Nil -> xs
    | Snoc (ys', y) -> Snoc (append' xs ys', y)
    | Cons (y, ys') -> append' (Snoc (xs, y)) ys'

// Es bringt jedoch keinen Vorteil, das Muster (xs, ys) abzugleichen, denn
// solange beide Folgen weder leer noch einelementig sind lässt sich der
// rekursive Aufruf nicht vermeiden. Insbesondere muss auf die Terminierung
// der Funktion geachten werden, wenn Elemente in beide Richtungen zwischen
// den beiden Folgen verschoben werden.
```

Aufgabe 5 Bäume (__ /20 Punkte)

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

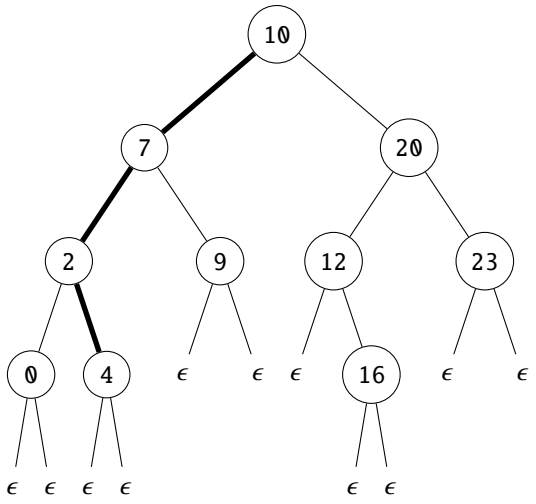
Wir betrachten folgende Typen:

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

type Path = | End | Left of Path | Right of Path
```

Bäume des Typs `Tree<'a>` speichern ihre Daten in den Knoten. Der Typ `Path` repräsentiert Pfade im Baum. Pfade beginnen immer in der Wurzel des Baumes.

Beispiel: Rechts sehen Sie den Code zu dem hier dargestellten Baum und dem hervorgehobenen Pfad mit den Knoten 10, 7, 2, 4.




```
let exTree: Tree<Nat> =
  Node (
    Node (
      Node (
        Node (Leaf, 0, Leaf),
        2,
        Node (Leaf, 4, Leaf)
      ),
      7,
      Node (Leaf, 9, Leaf)
    ),
    10,
    Node (
      Node (
        Leaf,
        12,
        Node (Leaf, 16, Leaf)
      ),
      20,
      Node (Leaf, 23, Leaf)
    )
  )

let exPath: Path = Left (Left (Right End))
```

a) Schreiben Sie eine Funktion `lookup: Tree<'a> -> Path -> Option<'a>`, die den Wert des Knotens bestimmt, zu dem der gegebenen Pfad führt. Wenn der Pfad zu keinem Knoten führt, soll `None` zurückgegeben werden.

Beispiele:

```
lookup exTree End = Some 10
lookup exTree exPath = Some 4
lookup exTree (Left (Right (Left End))) = None
lookup exTree (Left (Right (Left (Left End)))) = None
```

__ /6

```
let rec lookup (t: Tree<'a>) (p: Path): Option<'a> =
    match (t, p) with
    | (Leaf, _)                   -> None
    | (Node (_, x, _), End)       -> Some x
    | (Node (l, _, _), Left pp)   -> lookup l pp
    | (Node (_, _, r), Right pp)  -> lookup r pp
```

b) Schreiben Sie eine Funktion `update: Tree<'a> -> Path -> 'a -> Tree<'a>`, die den Baum berechnet, der entsteht, wenn man im gegebenen Baum den Wert des Knotens, zu dem der Pfad führt, zum gegebenen Wert `x` ändert. Führt der Pfad zu keinem Knoten, soll der Baum unverändert zurückgegeben werden.

Beispiele:

```
let t = Node (Node (Leaf, 2, Leaf), 3, Leaf)
update t End 4 = Node (Node (Leaf, 2, Leaf), 4, Leaf)
update t (Left End) 4 = Node (Node (Leaf, 4, Leaf), 3, Leaf)
update t (Right End) 4 = Node (Node (Leaf, 2, Leaf), 3, Leaf)
```

__/6

```
let rec update (t: Tree<'a>) (p: Path) (x: 'a): Tree<'a> =
    match (t, p) with
    | (Leaf, _)                  -> t
    | (Node (l, _, r), End)      -> Node (l, x, r)
    | (Node (l, y, r), Left pp)  -> Node (update l pp x, y, r)
    | (Node (l, y, r), Right pp) -> Node (l, y, update r pp x)
```

c) Schreiben Sie eine Funktion `search: Tree<Nat> -> Nat -> Option<Path>`, die im gegebenen *Suchbaum* einen Pfad zum gegebenen Wert `x` bestimmt. Befindet sich `x` nicht im Baum, soll `None` zurückgegeben werden.

Zur Erinnerung: Ein Suchbaum ist ein Baum, bei dem für jeden Knoten die Elemente im linken Teilbaum kleiner und im rechten Teilbaum größer sind als das Element im Knoten selbst. Der Baum `exTree` oben ist ein Suchbaum.

Nutzen Sie aus, dass es sich um einen Suchbaum handelt.

Beispiel:

```
search exTree 10 = Some End
search exTree 4 = Some (Left (Left (Right (End))))
search exTree 3 = None
```

__/8

```
let rec search (t: Tree<Nat>) (x: Nat): Option<Path> =
    match t with
    | Leaf -> None
    | Node (l, y, r) ->
        if x = y then Some End
        elif x < y then
            match search l x with
            | None -> None
            | Some p -> Some (Left p)
        else
            match search r x with
            | None -> None
            | Some p -> Some (Right p)
```

Aufgabe 6 Reguläre Ausdrücke (__ /20 Punkte)

a) Wir betrachten den regulären Ausdruck `x*(xyx)*x*`. 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.

__ /5

Wort	enthalten	nicht enthalten
xxxxxxxxxx	X	
xxxyxxyxxx	X	
xxyyxxyyxx		X
xyxxyxxyxx	X	
xxxyxxyxxy		X

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

__ /8

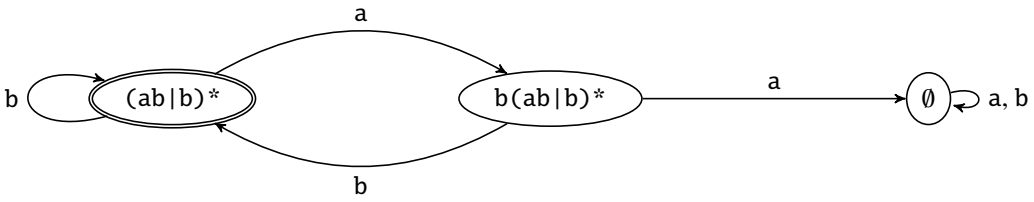
```
(ab|ba) / a = ((ab)/a) | ((ba)/a)
= b | ∅
= b

(a(b*))* / b = ((a(b*))/b) (a(b*))*
= ∅

((abc)*cba) / a = ((abc)*/a)cba | Δ((abc)*)((cba)/a)
= ((abc)/a)(abc)* cba | ϵ ∅
= bc(abc)*cba

((abc)*cba) / c = ((abc)*/c)cba | Δ((abc)*)((cba)/c)
= ∅ | ba
= ba
```

c) Für den regulären Ausdruck `(ab|b)*` wurden die Rechtsfaktoren, wie in folgendem Graphen dargestellt, ermittelt. Implementieren Sie daraus die Akzeptorfunktionen. Gehen Sie dabei **nach dem Verfahren aus der Vorlesung** vor. Nutzen Sie für das Alphabet den Typ `type Alphabet = | A | B`.



__/7

```
let rec accept0 (input: Alphabet list): Bool = // (ab|b)*
match input with
| [] -> true
| A::rest -> accept1 rest
| B::rest -> accept0 rest
and accept1 (input: Alphabet list): Bool = // b(ab|b)*
match input with
| [] -> false
| A::rest -> accept2 rest
| B::rest -> accept0 rest
and accept2 (input: Alphabet list): Bool = // ∅
match input with
| [] -> false
| A::rest -> accept2 rest
| B::rest -> accept2 rest
```