Aufgabe 1 Statische Semantik (__ /20 Punkte)

Füllen Sie die Lücken, sodass sich **gültige Aussagen der Statischen Semantik** ergeben. Alle vorkommenden Zahlen sind vom Typ `Nat`.

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

a) $\emptyset \vdash$ **false** : _____

$\emptyset \vdash$ **false** : Bool

b) $\emptyset \vdash$ _____ : Bool

$\emptyset \vdash$ **false** : Bool

c) $\emptyset \vdash$ **if** _____ **then** 4711 **else** _____ : _____

$\emptyset \vdash$ **if true then** 4711 **else** 815 : Nat

d) $\emptyset \vdash$ **let** x = _____ **in** x 4711 : Bool

$\emptyset \vdash$ **let** x = **fun** (n: Nat) -> **false in** x 4711 : Bool

e) $\emptyset \vdash$ _____ : Unit * Unit

$\emptyset \vdash$ ((),()) : Unit * Unit

f) $\emptyset \vdash$ **let rec** f (x: Nat): Bool = f x : _____

$\emptyset \vdash$ **let rec** f (x: Nat): Bool = f x : $\{$ f $\mapsto$ Nat $\rightarrow$ Bool $\}$

g) $\{$ _____ $\}$ $\vdash$ putline "Harry Hacker"; x : _____

$\{$ x $\mapsto$ Nat $\}$ $\vdash$ putline "Harry Hacker"; x : Nat

Aufgabe 2 Beweissysteme (__ /20 Punkte)

Wir betrachten folgende Baumsprache für Binärbäume:

$t \in \text{Tree} ::= \text{Leaf} \mid \text{Node}\ (\text{Tree}, \mathbb{N}, \text{Tree})$

a) Das Prädikat list−like beschreibt listenartige Binärbäume, bei denen jeder Knoten maximal einen Teilbaum hat:

$$\frac{}{\text{list−like Leaf}} \qquad \frac{\text{list−like } t}{\text{list−like (Node } (t, n, \text{Leaf}))} \qquad \frac{\text{list−like } t}{\text{list−like (Node (Leaf}, n, t))}$$

Prüfen Sie jeweils, ob die folgenden Binärbäume das Prädikat list−like erfüllen. Wenn ja, dann geben Sie einen **vollständigen Beweisbaum** an. Wenn nein, dann **begründen Sie kurz warum nicht**.

- Node (Leaf, 1, Node (Leaf, 2, Node (Leaf, 3, Leaf))) **__ /10**

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\text{list−like } \mathit{Leaf}}}{\text{list−like (Node (Leaf}, 3, \text{Leaf))}}}{\text{list−like (Node (Leaf}, 2, \text{Node (Leaf}, 3, \text{Leaf)))}}}{\text{list−like (Node (Leaf}, 1, \text{Node (Leaf}, 2, \text{Node (Leaf}, 3, \text{Leaf))))}}$$

- Node (Node (Leaf, 1, Leaf), 2, Node (Leaf, 3, Leaf))

Dieser Baum ist nicht list−like, da der Wurzelknoten zwei Node-Teilbäume hat, aber die Beweisregeln ein Leaf auf mindestens einer der beiden Seiten benötigen.

- Node (Leaf, 1, Node (Node (Leaf, 2, Leaf), 3, Leaf))

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\text{list−like } \mathit{Leaf}}}{\text{list−like (Node (Leaf}, 2, \text{Leaf))}}}{\text{list−like (Node (Node (Leaf}, 2, \text{Leaf)}, 3, \text{Leaf))}}}{\text{list−like (Node (Leaf}, 1, \text{Node (Node (Leaf}, 2, \text{Leaf)}, 3, \text{Leaf)))}}$$

b) Geben Sie ein Beweissystem für vollständige Binärbäume an. Definieren Sie dazu Beweisregeln für ein Prädikat

$$\text{perfect } h$$

für vollständige Binärbäume der Höhe h.

Beispiele für Binärbäume, die das Prädikat erfüllen: **__ /10**

```
perfect 0 Leaf
perfect 1 (Node (Leaf, 4711, Leaf))
perfect 2 (Node (Node (Leaf, 123, Leaf), 456, Node (Leaf, 789, Leaf)))
perfect 3 (Node (
   Node (Node (Leaf, 1, Leaf), 2, Node (Leaf, 3, Leaf)),
   4,
   Node (Node (Leaf, 5, Leaf), 6, Node (Leaf, 7, Leaf))
))
```

$$\frac{}{\text{perfect } 0 \text{ Leaf}} \qquad \frac{\text{perfect } h\ t_1 \qquad \text{perfect } h\ t_2}{\text{perfect } (h+1)\ (\text{Node } (t_1, n, t_2))}$$

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 `countIf: (Nat -> Bool) -> Nat -> Nat`, die ein Prädikat `p` und eine natürliche Zahl `n` nimmt und die Anzahl der Zahlen von `0` bis `n` zurückgibt, für die `p` gilt. Gehen Sie **strikt nach Peano Entwurfsmuster** vor.

Beispiele:

```
countIf (fun n -> n%2 = 0) 10 = 6
countIf (fun n -> n > 5)    7 = 2
countIf (fun n -> n < 10)   0 = 1
```

__ /5

```
let rec countIf (p: Nat -> Bool) (n: Nat): Nat =
    if n = 0N then
        if p 0N then 1N else 0N
    else
        countIf p (n - 1N) + (if p n then 1N else 0N)
```

b) Schreiben Sie eine Funktion `evenInterval: Nat -> List<Nat>`, die eine natürliche Zahl `n` nimmt und die Liste aller geraden Zahlen von `n` bis `0` zurückgibt (absteigend). Gehen Sie **strikt nach Peano Entwurfsmuster** vor.

Beispiele:

```
evenInterval 10 = [10; 8; 6; 4; 2; 0]        evenInterval 1 = [0]
evenInterval 7  = [6; 4; 2; 0]               evenInterval 0 = [0]
```

__ /5

```
let rec evenInterval (n: Nat): List<Nat> =
    if n = 0N then [0N]
    else (if n % 2N = 0N then [n] else []) @ evenInterval (n - 1N)
```

Für die nächsten beiden Teilaufgaben verwenden wir folgenden Typen für Binärbäume:

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

Darüber hinaus verwenden wir folgende Funktionsdefinition:

```
let foldTree<'a, 'b> (f: 'b -> 'a -> 'b -> 'b) (z: 'b): Tree<'a> -> 'b =
    let rec helper (t: Tree<'a>): 'b =
        match t with
        | Leaf -> z
        | Node (l, x, r) -> f (helper l) x (helper r)
    in helper
```

Die Funktion `foldTree` abstrahiert das Struktur-Entwurfsmuster für Bäume, ähnlich zu `peano-pattern` aus der Vorlesung. Sie nimmt eine Funktion `f` sowie einen Wert `z`. Der Wert `z` ist der Rückgabewert des Basisfalls (`Leaf`), während `f` die Rückgabewerte der rekursiven Aufrufe sowie den im Knoten enthaltenen Wert verarbeitet.

c) Schreiben Sie eine Funktion `numberOfLeaves<'a>: Tree<'a> -> Nat`, die die Anzahl der Blätter eines Baums zählt. Verwenden Sie hierfür die Funktion `foldTree`.

Beispiele:
```
numberOfLeaves Leaf = 1
numberOfLeaves (Node (Node (Leaf, 1, Leaf), 2, Leaf)) = 3
numberOfLeaves (Node (Node (Leaf, 1, Leaf), 2, Node (Leaf, 3, Leaf))) = 4
```

```
let numberOfLeaves<'a> : Tree<'a> -> Nat =
    foldTree (fun l _ r -> l + r) 1N
```

__/5

d) Schreiben Sie eine Funktion `any<'a>: ('a -> Bool) -> Tree<'a> -> Bool`, die ein Prädikat `p` und einen Baum `t` nimmt und überprüft, ob `p` für mindestens ein Element des Baums gilt. Verwenden Sie hierfür die Funktion `foldTree`.

Beispiele:
```
any (fun n -> n % 2 = 0) Leaf = false
any (fun n -> n % 2 = 0) (Node (Node (Leaf, 1, Leaf), 2, Leaf)) = true
any (fun n -> n % 2 = 0) (Node (Node (Leaf, 1, Leaf), 3, Leaf)) = false
```

```
let any<'a> (p: 'a -> Bool): Tree<'a> -> Bool =
    foldTree (fun l x r -> l || p x || r) false
```

__/5

Aufgabe 4 Sequenzen (__ /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 Typ für unendliche Sequenzen:

```
type Stream<'a> = Unit -> ('a * Stream<'a>)
```

Wenn Sie diese Aufgabe am Computer nachprogrammieren möchten, dann benötigen Sie wegen der Implementierungsdetails von F# einen etwas komplizierteren Typ:

```
type Stream<'a> = | S of (Unit -> ('a * Stream<'a>))
```

In den Lösungen der ersten beiden Teilaufgaben muss dann jeweils der Konstruktor `S` mit `let (S s) = s` entfernt werden und bei den letzten beiden Teilaufgaben muss der Konstruktor `S` im Ergebnis ergänzt werden.

a) Schreiben Sie eine Funktion `head<'a>: Stream<'a> -> 'a`, die das erste Element einer Sequenz ermittelt.

__ /6

```
let head<'a> (s: Stream<'a>): 'a =
    fst (s())
```

b) Schreiben Sie eine Funktion `map<'a, 'b>: ('a -> 'b) -> Stream<'a> -> Stream<'b>`, die eine Funktion auf alle Elemente der Sequenz anwendet und die Ergebnisse als Sequenz zurückgibt.

__ /7

```
let rec map<'a, 'b> (f: 'a -> 'b) (s: Stream<'a>): Stream<'b> =
    fun _ -> let (h, t) = s() in (f h, map f t)
```

c) Schreiben Sie eine Funktion `repeat<'a>: 'a -> Stream<'a>`, die eine Sequenz erstellt welche den gegebenen Wert unendlich wiederholt.

__ /7

```
let rec repeat<'a> (x: 'a): Stream<'a> =
    fun _ -> (x, repeat x)
```

Aufgabe 5 Dünne Bäume (__ /20 Punkte)

Sie dürfen zur Lösung dieser Aufgabe Bibliotheksfunktionen verwenden.

Gegeben ist der folgende Typ für Bäume:

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

Ein Knoten kann also ein Element vom Typ `'a` enthalten, muss es aber nicht.

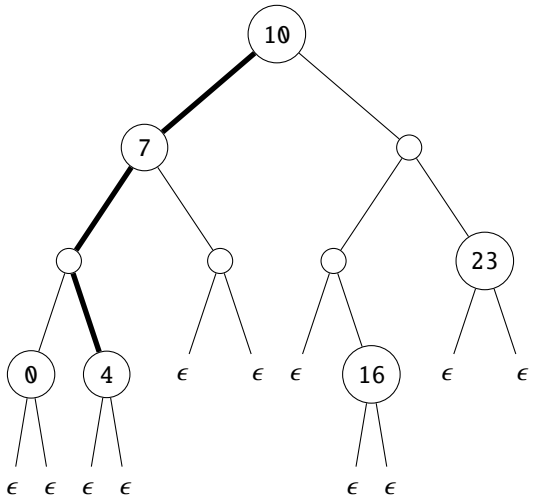
Darüber hinaus ist der folgende Typ für Pfade gegeben:

```
type Dir = | Left | Right

type Path = List<Dir>
```

Ein Pfad ist also eine Liste von Richtungen (`Dir`). Pfade beginnen immer in der Wurzel des Baums.

Beispiel: Rechts sehen Sie den Code zu dem hier dargestellten Baum und dem hervorgehobenen Pfad mit den Knoten `Some 10`, `Some 7`, `None`, `Some 4`.




```
let exTree: Tree<Nat> =
  Node (
    Node (
      Node (
        Node (Leaf, Some 0, Leaf),
        None,
        Node (Leaf, Some 4, Leaf)
      ),
      Some 7,
      Node (Leaf, None, Leaf)
    ),
    Some 10,
    Node (
      Node (
        Leaf,
        None,
        Node (Leaf, Some 16, Leaf)
      ),
      None,
      Node (Leaf, Some 23, Leaf)
    )
  )

let exPath: Path = [Left; Left; Right]
```

a) Schreiben Sie eine Funktion `toList<'a>: Tree<'a> -> List<'a * Path>`, die einen Baum in eine Liste von Knotenwerten mit den dazugehörigen Pfaden überführt. Die Reihenfolge der Elemente in der Liste ist dabei nicht relevant.

__/10

Beispiele:

```
toList Leaf = []
toList (Node (Leaf, Some 5, Leaf)) = [(5, [])]
toList (Node (Leaf, None, Leaf)) = []
toList exTree = [(0, [Left; Left; Left]); (4, [Left; Left; Right]); (7, [Left]);
                 (10, []); (16, [Right; Left; Right]); (23, [Right; Right])]
```

```
let rec toList<'a> (t: Tree<'a>): List<'a * Path> =
  let addDir d = List.map (fun (x, p) -> (x, d :: p))
  match t with
  | Leaf           -> []
  | Node (l, x, r) -> match x with
                      | None   -> []
                      | Some x -> [(x, [])]
                      @ addDir Left  (toList l)
                      @ addDir Right (toList r)

// Alternativ: Pfad in zusätzlichem Parameter aufbauen
let toList'<'a> (t: Tree<'a>): List<'a * Path> =
  let rec h (t: Tree<'a>) (p: Path): List<'a * Path> =
    match t with
    | Leaf -> []
    | Node (l, x, r) -> match x with
                        | None   -> []
                        | Some x -> [(x, p)]
                        @ h l (p @ [Left])
                        @ h r (p @ [Right])
  h t []
```

b) Schreiben Sie eine Funktion `fromList<'a>: List<'a * Path> -> Tree<'a>`, die eine Liste von Paaren von Werten und Pfaden nimmt und einen Baum zurückgibt, der an den Stellen der Pfade die Werte enthält. Alle anderen Knoten des Baums sind `None`. Sie dürfen davon ausgehen, dass jeder Pfad nur einmal in der Liste enthalten ist. **Erklären Sie kurz Ihren Lösungsweg in ein bis zwei Sätzen, zum Beispiel als Kommentar im Code**.

__/10

Beispiele:

```
fromList [] = Leaf
fromList [(5, [])] = Node (Leaf, Some 5, Leaf)
fromList [(5, [Left]); (6, [])] = Node (Node (Leaf, Some 5, Leaf), Some 6, Leaf)
```

Erster Lösungsweg: Nach der Struktur der Eingabeliste Wir starten mit einem Leaf und fügen nacheinander die einzelnen Einträge aus der Liste in den Baum ein. Immer wenn wir beim Einfügen in den Baum auf ein Leaf treffen, erweitern wir den Baum an dieser Stelle vorerst um ein Node ohne Element und mit leeren Teilbäumen. Die Einfügeoperation wiederum arbeitet rekursiv auf dem durch das erste Pfadelement vorgegebenen Teilbaum.

```
let fromList<'a> (xs: List<'a * Path>): Tree<'a> =
  let rec insert (tree: Tree<'a>) (elem: 'a, path: Path): Tree<'a> =
    match tree with
    | Leaf           -> insert (Node (Leaf, None, Leaf)) (elem, path)
    | Node (l, x, r) -> match path with
                        | []         -> Node (l, Some elem, r)
                        | Left  :: p -> Node (insert l (elem, p), x, r)
                        | Right :: p -> Node (l, x, insert r (elem, p))
  List.fold insert Leaf xs
```

Zweiter Lösungsweg: Nach der Struktur des zu erstellenden Baumes Wenn die Eingabeliste nicht leer ist, dann splitten wir sie in drei Teile, nämlich eine Teilliste für alle Einträge deren Pfad mit `Left` beginnt, eine Teilliste für alle Einträge deren Pfad mit `Right` beginnt, sowie das optionale Element mit dem leeren Pfad für den Wurzelknoten. Bei den Teillisten entfernen wir dabei jeweils das erste Pfadelement, sodass wir sie in einem rekursiven Aufruf wieder an `fromList` übergeben können um die beiden Teilbäume zu erhalten.

```
let rec fromList'<'a> (xs: List<'a * Path>): Tree<'a> =
  match xs with
  | [] -> Leaf
  | _  -> let mutable root   = None
          let mutable lefts  = []
          let mutable rights = []
          for (elem, path) in xs do
            match path with
            | []         -> root   <- Some elem
            | Left  :: p -> lefts  <- (elem, p) :: lefts
            | Right :: p -> rights <- (elem, p) :: rights
          Node (fromList' lefts, root, fromList' rights)
```

Aufgabe 6 Reguläre Ausdrücke (__ /20 Punkte)

a) Wir betrachten den regulären Ausdruck

$$\mathsf{a} \cdot \mathsf{b} \cdot \Big((\mathsf{b} \cdot \mathsf{a} \cdot \mathsf{b}) \mid (\mathsf{a} \cdot \mathsf{b} \cdot \mathsf{a}) \mid (\mathsf{a} \cdot \mathsf{a})\Big)^* \cdot \mathsf{b}$$

über dem Alphabet $\{\mathsf{a}, \mathsf{b}\}$.

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
bababbbababba		X
ababaaababaab	X	
abbababaaaaab	X	
abababbabaaab		X
abaaaaaaaaaab	X	
abababababbb		X

b) Wir erweitern die Syntax für Reguläre Ausdrücke: $r \in \text{Reg} ::= \ldots$ bisherige Definitionen
$\mid \; r^n$ n-fache Wiederholung

Die Reduktionssemantik ist wie folgt definiert: $\dfrac{}{r^0 \to \epsilon}$ $\qquad$ $\dfrac{}{r^{n+1} \to r \cdot r^n}$

Erweitern Sie die Definition von *nullable* und die Regeln für Rechtsfaktoren um n-fache Wiederholungen. Sie müssen nur die neu hinzu gekommenen Definitionen aufschreiben, nicht die bisherigen Definitionen.

__ /6

$$\begin{aligned} \mathit{nullable}(r^0) &= \mathit{true} & x \setminus r^0 &= \emptyset \\ \mathit{nullable}(r^{n+1}) &= \mathit{nullable}(r) & x \setminus r^{n+1} &= (x \setminus r) \cdot r^n \end{aligned}$$

c) Wir verwenden die in der vorherigen Teilaufgabe eingeführte n-fache Wiederholung und betrachten folgenden regulären Ausdruck:

$$\mathsf{b}^* \cdot (\mathsf{a} \cdot \mathsf{b})^{4711}$$

i) Welche Sprache definiert dieser reguläre Ausdruck? Geben Sie eine umgangssprachliche Beschreibung an.

Die Sprache der Wörter, **__/2**

- die mit beliebig vielen b beginnen
- und dann genau 4711 mal die Buchstaben ab folgen.

ii) Wie viele Rechtsfaktoren hat der reguläre Ausdruck?

Es sind $2 * 4711 + 2 =$ **9424** Rechtsfaktoren. **__/2**
Es war nicht verlangt, die genaue Auflistung und Herleitung anzugeben, der Vollständigkeit halber ist sie hier dennoch. Für $1 \leq n \leq 4711$:

$$\begin{aligned}
r_0 &:= \mathsf{b}^* \cdot (\mathsf{a} \cdot \mathsf{b})^{4711} \\
r_{2n-1} &:= \mathsf{b} \cdot (\mathsf{a} \cdot \mathsf{b})^{4711-n} \\
r_{2n} &:= (\mathsf{a} \cdot \mathsf{b})^{4711-n} \\
r_{9423} &:= \emptyset
\end{aligned}$$

mit

$$\begin{aligned}
\mathsf{a} \setminus r_0 &= r_1 \\
\mathsf{b} \setminus r_0 &= r_0 \\
\mathsf{a} \setminus r_{2n-1} &= r_{9423} \\
\mathsf{b} \setminus r_{2n-1} &= r_{2n} \\
\mathsf{a} \setminus r_{2n} &= r_{2n+1} \\
\mathsf{b} \setminus r_{2n} &= r_{9423} \\
\mathsf{a} \setminus r_{9423} &= r_{9423} \\
\mathsf{b} \setminus r_{9423} &= r_{9423}
\end{aligned}$$

iii) Schreiben Sie eine Akzeptorfunktion (nicht zwingend nach dem Schema aus der Vorlesung).

```
type Alphabet = | A | B
```

__/4

```
let rec accept (xs: List<Alphabet>): Bool =
  match xs with
  | []    -> false
  | A::xs -> accept1 1 xs
  | B::xs -> accept xs

and accept1 (n: Nat) (xs: List<Alphabet>): Bool =
  match xs with
  | B::xs -> accept2 n xs
  | _     -> false

and accept2 (n: Nat) (xs: List<Alphabet>): Bool =
  match xs with
  | []    -> n = 4711
  | A::xs -> accept1 (n + 1) xs
  | _     -> false


// Alternativ, weniger nah am Schema aus der Vorlesung, dafür kürzer:
let rec accept' (xs: List<Alphabet>): Bool =
  match xs with
  | B   ::xs -> accept'    xs
  | A::B::xs -> accept1' 1 xs
  | _        -> false

and accept1' (n: Nat) (xs: List<Alphabet>): Bool =
  match xs with
  | []       -> n = 4711
  | A::B::xs -> accept1' (n + 1) xs
  | _        -> false
```

Aufgabe 7 REPL (__ /20 Punkte)

Sie dürfen zur Lösung dieser Aufgabe Bibliotheksfunktionen verwenden.

In dieser Aufgabe sollen Sie eine einen Aufzähler in Form einer REPL (Read-Eval-Print-Loop) implementieren. Der Ablauf des Programms ist wie folgt: Der Benutzer wird durch ein `>␣Enter␣a␣number:␣` aufgefordert, eine Zahl einzugeben. Nachdem der Benutzer eine Zahl eingegeben hat, wird die Zahl auf die bisherige Summe addiert und die neue Summe mit `>␣The␣sum␣is:␣` ausgegeben. Anschließend wird der Benutzer erneut aufgefordert, eine Zahl einzugeben. Der Benutzer kann das Programm durch Eingabe von `x` beenden. In diesem Fall wird die Summe mit `>␣Final␣sum:␣` ausgegeben und das Programm beendet. Ist die Eingabe keine Zahl, so wird eine Fehlermeldung `>␣Invalid␣input.␣Please␣enter␣a␣number.` ausgegeben und der Benutzer erneut aufgefordert, eine Zahl einzugeben.

Beispielinteraktion:

```
> Enter a number: 5
The sum is: 5
> Enter a number: 3
The sum is: 8
> Enter a number: keineZahl
Invalid input. Please enter a number.
> Enter a number: 4
The sum is: 12
> Enter a number: x
Final sum: 12
```

Hinweis: Die Funktion `readNat: String -> Nat` konvertiert einen String in eine natürliche Zahl. Sollte die Eingabe keine Zahl sein, so wird eine `Read`-Ausnahme geworfen.

a) Schreiben Sie eine Funktion `replRec: Unit -> Unit`, die die oben beschriebene REPL funktional implementiert. Die Lösung darf **weder** Schleifen **noch** mutable Variablen bzw. Referenzen verwenden.

__/10

```
let replRec (): Unit =
  let rec loop (sum: Nat): Unit =
    putstring "> Enter a number: "
    let input = getline ()
    if input = "x" then putline ("Final sum: " ^ (show sum))
    else
      try
        let n = readNat input
        let sum' = sum + n
        putline ("Current sum: " ^ (show sum'))
        loop sum'
      with
      | Read -> putline "Invalid input. Please enter a number." ; loop sum
  loop 0N
```

b) Schreiben Sie eine Funktion `replImp: Unit -> Unit`, die die oben beschriebene REPL imperativ implementiert. Die Lösung darf **keine** rekursive Funktion verwenden.

__/10

```
let replImp (): Unit =
  let mutable sum = 0N
  let mutable cont = true
  while cont do
    putstring "> Enter a number: "
    let input = getline ()
    if input = "x" then
      putline ("Final sum: " ^ (show sum))
      cont <- false
    else
      try
        let n = readNat input
        sum <- sum + n
        putline ("Current sum: " ^ (show sum))
      with
      | Read -> putline "Invalid input. Please enter a number."
```

Aufgabe 8 Ausnahmen (__ /20 Punkte)

a) Unter Berücksichtigung dieser Typ- und Ausnahmedefinitionen

```
type AB = | A of Nat | B

exception E of Nat
exception F of Bool
```

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

```
try
    match f() with
    | A x -> if x < 10 then raise (E x) else x
    | B   -> raise (F true)
with
| E x -> if x = 0 then raise (F false) else x
| F x -> if x then 4711 else raise (E 7)
```

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

1. `let f() = A 0` **__ /10**

   `F false`

2. `let f() = A 4711`

   `4711`

3. `let f() = B`

   `4711`

4. `let f() = raise (E 99)`

   `99`

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

   `E 7`

b) Wir betrachten folgende Implementierung für eine Warteschlange:

```
1  let buffer = [| ref 0; ref 0; ref 0; ref 0 |]
2  let writePos = ref 0
3  let size = ref 0
4
5  let enqueue (x: Nat): Unit =
6    buffer.[!writePos] := x
7    size := !size + 1
8    writePos := (!writePos + 1) % buffer.Length
9
10 let dequeue (): Nat =
11   let readPos = (!writePos - !size + buffer.Length) % buffer.Length
12   size := !size - 1
13   !buffer.[readPos]
```

Allerdings funktioniert die Warteschlange nicht wie erwartet, wie die folgenden beiden Beispiele zeigen (die Beispiele beginnen jeweils mit dem Ursprungszustand):

```
enqueue 1
enqueue 2
enqueue 3
enqueue 4
enqueue 5
enqueue 6
dequeue () // gibt 5 zurück
```

```
enqueue 1
enqueue 2
dequeue () // gibt 1 zurück
dequeue () // gibt 2 zurück
dequeue () // gibt 0 zurück
dequeue () // gibt 0 zurück
dequeue () // gibt 1 zurück
```

Definieren Sie geeignete Ausnahmen und passen Sie die Implementierung so an, dass statt des fehlerhaften Verhaltens die Ausnahmen geworfen werden. Sie müssen die korrigierte Implementierung nicht vollständig aufschreiben, geben Sie nur den zu ändernden Code an und verweisen Sie auf die Zeilennummern, wo der Code einzufügen ist.

Füge vor Zeile 1 ein: **__ / 10**

```
exception BufferFull
exception BufferEmpty
```

Füge zwischen Zeile 5 und 6 ein:

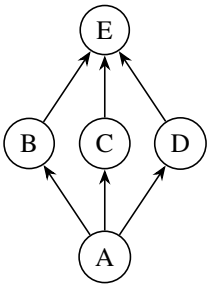
```
  if (!size = buffer.Length) then raise BufferFull
```

Füge zwischen Zeile 10 und 11 ein:

```
  if (!size = 0) then raise BufferEmpty
```

Aufgabe 9 Untertypen (__ /20 Punkte)

Es gelten die Untertypbeziehungen A $\preccurlyeq$ B, A $\preccurlyeq$ C, A $\preccurlyeq$ D, B $\preccurlyeq$ E, 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 $\parallel$). In Lücken, die Kästen ( ) enthalten, müssen Sie in jeden Kasten einen der Typen A, B, C, D oder E eintragen.

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	$\boxed{\parallel}$	D
E $*$ A	$\prec$	$\boxed{\text{E} * \text{B}}$, $\boxed{\text{E} * \text{C}}$, $\boxed{\text{E} * \text{D}}$, $\boxed{\text{E} * \text{E}}$
A $*$ E	$\prec$	$\boxed{\text{B} * \text{E}}$, $\boxed{\text{C} * \text{E}}$, $\boxed{\text{D} * \text{E}}$, $\boxed{\text{E} * \text{E}}$
C $\to$ D	$\prec$	$\boxed{\text{A} \to \text{D}}$, $\boxed{\text{A} \to \text{E}}$, $\boxed{\text{C} \to \text{E}}$
C $\to$ C	$\prec$	$\boxed{\text{A} \to \text{C}}$, $\boxed{\text{A} \to \text{E}}$, $\boxed{\text{C} \to \text{E}}$
E $\to$ A	$\boxed{\prec}$	A $\to$ E
B $\to$ D	$\boxed{\succ}$	E $\to$ A
B $\to$ C $\to$ D	$\prec$	$\boxed{\text{A} \to \text{A} \to \text{D}}$, $\boxed{\text{A} \to \text{A} \to \text{E}}$, $\boxed{\text{A} \to \text{C} \to \text{D}}$, $\boxed{\text{A} \to \text{C} \to \text{E}}$ $\boxed{\text{B} \to \text{A} \to \text{D}}$, $\boxed{\text{B} \to \text{A} \to \text{E}}$, $\boxed{\text{B} \to \text{C} \to \text{E}}$
B $\to$ C $\to$ D	$\parallel$	Es gibt **110 richtige Lösungen**, eine davon ist: $\boxed{\text{A} \to \text{A} \to \text{A}}$ **Falsch sind** die in der vorherigen Zeile genannten ($\prec$) sowie die folgenden ($\succeq$): $\boxed{\text{B} \to \text{C} \to \text{A}}$, $\boxed{\text{B} \to \text{C} \to \text{D}}$, $\boxed{\text{B} \to \text{E} \to \text{A}}$, $\boxed{\text{B} \to \text{E} \to \text{D}}$ $\boxed{\text{E} \to \text{C} \to \text{A}}$, $\boxed{\text{E} \to \text{C} \to \text{D}}$, $\boxed{\text{E} \to \text{E} \to \text{A}}$, $\boxed{\text{E} \to \text{E} \to \text{D}}$
(B $\to$ C) $\to$ D	$\prec$	$\boxed{(\text{B} \to \text{A}) \to \text{D}}$, $\boxed{(\text{B} \to \text{A}) \to \text{E}}$, $\boxed{(\text{B} \to \text{C}) \to \text{E}}$ $\boxed{(\text{E} \to \text{A}) \to \text{D}}$, $\boxed{(\text{E} \to \text{A}) \to \text{E}}$, $\boxed{(\text{E} \to \text{C}) \to \text{D}}$, $\boxed{(\text{E} \to \text{C}) \to \text{E}}$

Aufgabe 10 Funktional versus objektorientiert (__ / 20 Punkte)

Bäume lassen sich sowohl funktional als auch objektorientiert modellieren. Im Folgenden sind Binärbäume sowie eine Operation zum Zählen der Knoten definiert.

Funktionale Modellierung:

```
type Tree = | Leaf
            | Node of (Tree * Tree)

let rec countNodes (tree: Tree): Nat =
    match tree with
    | Leaf               -> 0
    | Node (left, right) -> 1 + countNodes left + countNodes right
```

Objektorientierte Modellierung:

```
type ITree =
    interface
        abstract member CountNodes: Nat
    end

let leaf (): ITree =
    { new ITree with
        member self.CountNodes = 0
    }

let node (left: ITree, right: ITree): ITree =
    { new ITree with
        member self.CountNodes = 1 + left.CountNodes + right.CountNodes
    }
```

In den folgenden Teilaufgaben soll das Modell der Bäume erweitert werden. Beachten Sie, dass die Teilaufgaben **unabhängig** voneinander sind. Die in Teilaufgabe a) hinzugefügte Operation ist *nicht* Bestandteil von Teilaufgabe b).

a) Erweitern Sie **ein Modell Ihrer Wahl** um eine neue Operation, die die Höhe des Baumes berechnet. Geben Sie an, welches Modell Sie erweitern. Verändern Sie dabei *nicht* die vorhandenen Definitionen.

__/10

```
// Erweiterung der funktionalen Modellierung
let rec height (tree: Tree): Nat =
    match tree with
    | Leaf               -> 0
    | Node (left, right) -> 1 + max (height left) (height right)
```

b) Erweitern Sie **ein Modell Ihrer Wahl** um eine neue Art von Knoten, der drei Teilbäume hat. Geben Sie an, welches Modell Sie erweitern. Verändern Sie dabei *nicht* die vorhandenen Definitionen.

__/10

```
// Erweiterung der objektorientierten Modellierung
let node3 (left: ITree) (middle: ITree) (right: ITree): ITree =
    { new ITree with
        member self.CountNodes =
          1 + left.CountNodes + middle.CountNodes + right.CountNodes
    }
```