

# Modulprüfung Grundlagen der Programmierung

Donnerstag, 07.03.2024

Exemplar-ID: 4711

|                 |  |
|-----------------|--|
| Nachname:       |  |
| Vorname:        |  |
| Matrikelnummer: |  |

## Hinweise:

1. Schreiben Sie **direkt bei Beginn** der Klausur Ihren Namen, Vornamen und Matrikelnummer **auf dieses Deckblatt**.
2. Achten Sie darauf, dass Ihre Klausur vollständig ist (21 Seiten)!
3. Sie haben 150 Minuten Zeit, die Klausur zu bearbeiten.
4. Schreiben Sie Ihre Lösungen gut lesbar mit Kugelschreiber oder Füllfederhalter (**kein Bleistift, kein Rotstift, kein Grünstift**)! Unleserliche Lösungen werden nicht korrigiert!
5. Sie dürfen **keine** eigenen Blätter verwenden. Lassen Sie diese Klausur in Ihrem eigenen Interesse geheftet; lose Klausurblätter werden nicht korrigiert!
6. Die Aufgaben **müssen** auf den jeweiligen Blättern bearbeitet werden. Sollte der Platz nicht ausreichen, so benutzen Sie die Rückseite des betreffenden Blattes oder die Zusatzblätter am Ende der Klausur. Sollte auch dies nicht ausreichen, bekommen Sie weitere Blätter bei der Aufsicht. Verweisen Sie in jedem Fall deutlich auf die Fortsetzungen Ihrer Aufgaben!
7. **Als Hilfsmittel zur Klausur zugelassen sind zwei beidseitig handbeschriebene A4-Blätter sowie Sprachwörterbücher.** Darüber hinaus sind keine Hilfsmittel erlaubt. Die Benutzung von Handys, Smartwatches und anderen elektronischen Geräten ist nicht gestattet. Handys müssen ausgeschaltet sein! Auf Ihrem Platz darf sich kein Rucksack o. ä. befinden. **Bei Verstößen gegen diese Regeln sowie bei Täuschungsversuchen wird die Klausur mit 0 Punkten gewertet. Täuschungsversuche werden darüber hinaus dem Prüfungsamt gemeldet.**
8. Lesen Sie vor der Bearbeitung einer Aufgabe den gesamten Aufgabentext sorgfältig durch! Die Aufgabenteile jeder Aufgabe bauen in der Regel nicht aufeinander auf. Sie können also in den meisten Fällen die Bearbeitung einer Aufgabe fortsetzen, auch wenn Sie einen Aufgabenteil nicht gelöst haben.
9. Der Hinweis "*Verwenden Sie keine Bibliotheksfunktionen*", mit dem einige Aufgaben versehen sind, bezieht sich auf F# Funktionen, die nach dem Schema `Modulname.funktionsname` benannt sind, also z.B. `List.map`. Die vordefinierten Funktionen `not`, `min`, `max` und `@` sind von diesem Verbot nicht betroffen.

| Aufgabe: | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 |
|----------|----|----|----|----|----|----|----|----|----|----|
| Punkte:  |    |    |    |    |    |    |    |    |    |    |
| Maximum: | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 |

|                  |     |
|------------------|-----|
| Gesamtpunktzahl: |     |
| Maximum:         | 200 |

**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`.

a)  $\emptyset \vdash \mathbf{false} : \underline{\hspace{2cm}}$ b)  $\emptyset \vdash \underline{\hspace{2cm}} : \mathbf{Bool}$ c)  $\emptyset \vdash \mathbf{if} \underline{\hspace{2cm}} \mathbf{then} 4711 \mathbf{else} \underline{\hspace{2cm}} : \underline{\hspace{2cm}}$ d)  $\emptyset \vdash \mathbf{let} \ x = \underline{\hspace{2cm}} \mathbf{in} \ x \ 4711 : \mathbf{Bool}$ e)  $\emptyset \vdash \underline{\hspace{2cm}} : \mathbf{Unit} * \mathbf{Unit}$ f)  $\emptyset \vdash \mathbf{let} \ \mathbf{rec} \ f \ (x: \mathbf{Nat}): \mathbf{Bool} = f \ x : \underline{\hspace{2cm}}$ g)  $\{ \underline{\hspace{2cm}} \} \vdash \mathbf{putline} \ "Harry \ Hacker"; \ x : \underline{\hspace{2cm}}$

**Aufgabe 2 Beweissysteme****( \_\_\_ / 20 Punkte)**

Wir betrachten folgende Baumsprache für Binärbäume:

$$t \in \text{Tree} ::= \text{Leaf}$$

$$| \text{Node}(\text{Tree}, \mathbb{N}, \text{Tree})$$

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

$$\frac{}{\text{list-like Leaf}} \quad \frac{\text{list-like } t}{\text{list-like}(\text{Node}(t, n, \text{Leaf}))} \quad \frac{\text{list-like } t}{\text{list-like}(\text{Node}(\text{Leaf}, n, t))}$$

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**

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

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

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))

))

\_\_\_ / 10

### 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
```

```
let rec countIf (p: Nat -> Bool) (n: Nat): Nat =
```

\_\_ / 5

- 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 7 = [6; 4; 2; 0]
```

```
evenInterval 1 = [0]
```

```
evenInterval 0 = [0]
```

```
let rec evenInterval (n: Nat): List<Nat> =
```

\_\_ / 5

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 =
```

\_\_\_/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 =
```

\_\_\_/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>)
```

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

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

\_\_/6

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.

```
let rec map<'a, 'b> (f: 'a -> 'b) (s: Stream<'a>): Stream<'b> =
```

\_\_/7

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

```
let rec repeat<'a> (x: 'a): Stream<'a> =
```

\_\_/7

**Aufgabe 5 Dünne Bäume****(\_\_/20 Punkte)****Sie dürfen zur Lösung dieser Aufgabe Bibliotheksfunktionen verwenden.**

Gegeben ist der folgende Typ für Bäume:

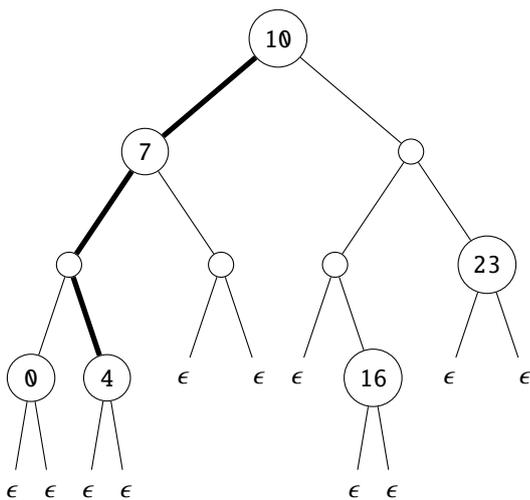
**type** Tree<'a> = | Leaf | Node **of** Tree<'a> \* Option<'a> \* Tree<'a>

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.

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> =
```

\_\_\_/10

- 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.**

Beispiele:

```
fromList [] = Leaf
```

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

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

```
let rec fromList<'a> (l: List<'a * Path>): Tree<'a> =
```

\_\_\_/10

## Aufgabe 6 Reguläre Ausdrücke

(\_\_ / 20 Punkte)

a) Wir betrachten den regulären Ausdruck

$$a \cdot b \cdot ((b \cdot a \cdot b) | (a \cdot b \cdot a) | (a \cdot a))^* \cdot b$$

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.

| Wort           | enthalten | nicht enthalten |
|----------------|-----------|-----------------|
| bababbbababba  |           |                 |
| ababaaababaab  |           |                 |
| abbababaaaaab  |           |                 |
| abababbabaaaab |           |                 |
| abaaaaaaaaaab  |           |                 |
| ababababababb  |           |                 |

\_\_ / 6

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

Die Reduktionssemantik ist wie folgt definiert:  $\frac{}{r^0 \rightarrow \epsilon}$   $\frac{}{r^{n+1} \rightarrow r \cdot r^n}$

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

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

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

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

\_\_\_/2

ii) Wie viele Rechtsfaktoren hat der reguläre Ausdruck?

\_\_\_/2

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

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

```
let rec accept (xs: List<Alphabet>): Bool =
```

\_\_\_/4

## 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.

**let** replRec (): Unit =

\_\_\_ / 10

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

**let** replImp () : Unit =

\_\_\_/10

**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 ausgewertet. 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

2. `let f() = A 4711`3. `let f() = B`4. `let f() = raise (E 99)`5. `let f() = raise (F false)`

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):

|  |  |
|--|--|
| <pre> enqueue 1 enqueue 2 enqueue 3 enqueue 4 enqueue 5 enqueue 6 dequeue () // gibt 5 zurück </pre> | <pre> enqueue 1 enqueue 2 dequeue () // gibt 1 zurück dequeue () // gibt 2 zurück dequeue () // gibt 0 zurück dequeue () // gibt 0 zurück dequeue () // gibt 1 zurück </pre> |
|--|--|

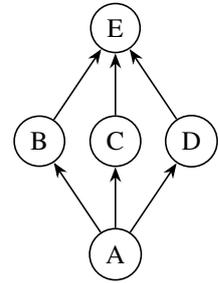
Definieren Sie geeignete Ausnahmen und passen Sie die Implementierung so an, dass statt des fehlerhaften Verhaltens die Ausnahmen geworfen werden. Sie müssen die korrigierte Implementierung nicht vollständig aufschreiben, geben Sie nur den zu ändernden Code an und verweisen Sie auf die Zeilennummern,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

\_\_\_/10

### 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:

- $t_1 < t_2$  bedeutet, dass  $t_1$  ein Untertyp von  $t_2$  ist ( $t_1 \preccurlyeq t_2$ ), nicht jedoch  $t_2$  ein Untertyp von  $t_1$ .
- $t_1 > t_2$  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 ( $<$  oder  $>$  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  $\preccurlyeq$ :

$$\frac{t_1 \preccurlyeq t_2 \quad t_2 \preccurlyeq t_3}{t_1 \preccurlyeq t_3}$$

$$\frac{t_1 \preccurlyeq t'_1 \quad t_2 \preccurlyeq t'_2}{t_1 * t_2 \preccurlyeq t'_1 * t'_2}$$

$$\frac{t'_1 \preccurlyeq t_1 \quad t_2 \preccurlyeq t'_2}{t_1 \rightarrow t_2 \preccurlyeq t'_1 \rightarrow t'_2}$$

| $t_1$                             | Relation    | $t_2$  |
|-----------------------------------|-------------|--|
| C                                 |             | D  |
| $E * A$                           | $<$         |  |
| $A * E$                           | $<$         |  |
| $C \rightarrow D$                 | $<$         |  |
| $C \rightarrow C$                 | $<$         |  |
| $E \rightarrow A$                 |             | $A \rightarrow E$  |
| $B \rightarrow D$                 |             | $E \rightarrow A$  |
| $B \rightarrow C \rightarrow D$   | $<$         |  |
| $B \rightarrow C \rightarrow D$   | $\parallel$ | <input type="text"/> $\rightarrow$ <input type="text"/> $\rightarrow$ <input type="text"/> |
| $(B \rightarrow C) \rightarrow D$ | $<$         |  |

---

**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

- 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

**Fortsetzung von Aufgabe \_\_\_\_\_**

**Fortsetzung von Aufgabe \_\_\_\_\_**