

Modulprüfung Konzepte der Programmierung

Donnerstag, 05.03.2026

Exemplar-ID: 180

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 (23 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)**

- 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!

__ / 10

i) $\emptyset \vdash 815 : \underline{\hspace{2cm}}$

ii) $\emptyset \vdash \text{if false then } \underline{\hspace{2cm}} \text{ else } \underline{\hspace{2cm}} : \text{Bool}$

iii) $\emptyset \vdash (4711, (\text{if } \underline{\hspace{2cm}} \text{ then } \underline{\hspace{2cm}} \text{ else } \underline{\hspace{2cm}})) : \text{Nat} * \text{Nat}$

iv) $\emptyset \vdash \text{let } x = 1 \text{ in } x < \underline{\hspace{2cm}} : \underline{\hspace{2cm}}$

v) $\{x \mapsto \underline{\hspace{2cm}}\} \vdash \text{fun } (y: \underline{\hspace{2cm}}) \rightarrow x + y : \underline{\hspace{2cm}}$

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

___/10

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

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!

__ / 10

i) $\{p \mapsto (4711, 815)\} \vdash \text{if } \underline{\hspace{2cm}} \text{ then fst } p \text{ else (fst } p * \text{snd } p) \Downarrow \underline{\hspace{2cm}}$

ii) $\underline{\hspace{2cm}} \vdash \text{let } f \text{ (x: Nat): Nat = x + a } \Downarrow \underline{\hspace{2cm}}$

iii) $\{x \mapsto 4711, f \mapsto \langle \{x \mapsto 42\}, x, x \rangle\} \vdash x \Downarrow \underline{\hspace{2cm}}$

iv) $\emptyset \vdash (\text{fun } x \rightarrow x \ 5) \ (\text{fun } y \rightarrow y + y) \Downarrow \underline{\hspace{2cm}}$

v) $\{x \mapsto 4711\} \vdash x > \underline{\hspace{2cm}} \Downarrow \text{true}$

- 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`

`let rec identity (n: Nat): Nat =`

__ / 5

- 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)`

`let rec evenOddSums (xs: List<Nat>): Nat * Nat =`

__ / 5

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

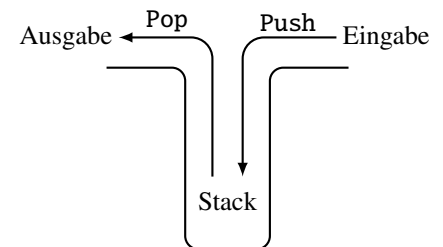
1. Ist die Funktion nach dem Peano-Entwurfsmuster geschrieben?
2. Für welche Eingabewerte terminiert die Funktion? Was ist der Rückgabewert im Fall der Termination?
3. Angenommen, die Funktion terminiert für den Eingabewert n . Wie viele Rekursionsschritte werden verwendet, um den Rückgabewert zu berechnen?
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.
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))

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.

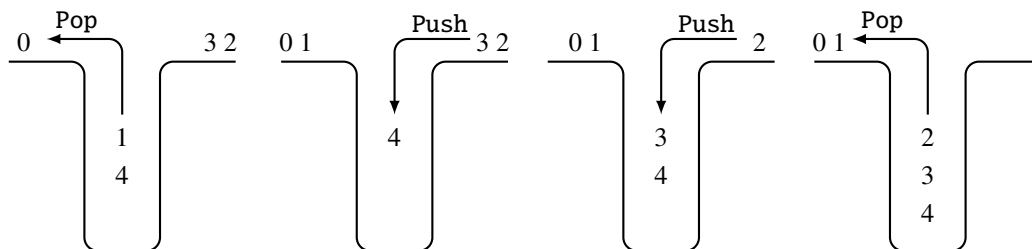


Die umgekehrte Folge `[2; 1; 0]` ist ebenfalls stack-sortierbar: `[Push; Push; Push; Pop; Pop; Pop]` sortiert die Eingabe – wir pushen zuerst alle Elemente und poppen sie dann. Nicht jede Permutation ist stack-sortierbar: `[1; 2; 0]` ist das kürzeste Gegenbeispiel.

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]`

type `Instruction = Push | Pop`

let rec `assemble (input: List<Nat>): List<Instruction> =`

Aufgabe 5 Ternärbäume

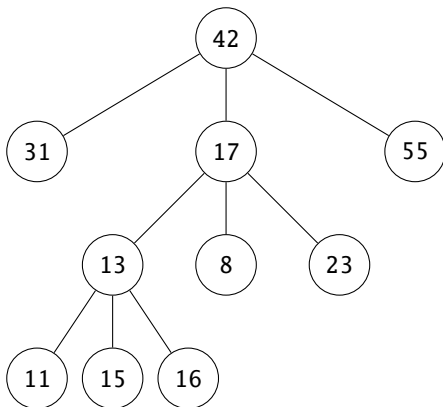
(__ / 20 Punkte)

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 (Leafs) 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
```

```
let rec size<'a> (t: Tree<'a>): Nat =
```

___/5

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))
```

```
let rec mirror<'a> (t: Tree<'a>): Tree<'a> =
```

___/5

- 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)]
```

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

___/5

- 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]
```

let `allOfSize (n: Nat): List<Tree<Unit> =`

___/5

Aufgabe 6 Reguläre Ausdrücke

(__ / 20 Punkte)

a) Wir betrachten den regulären Ausdruck

$$(a \cdot b)^* \cdot b \cdot \left((a \cdot a \cdot b) \mid (b \cdot a \cdot a) \right)^* \cdot a$$

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
ababababababa		
abbaabbaabbaa		
baabbaabaaaab		
bbbbbbbbbbba		
abbbaaaabaaba		
ababababbbaaa		

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 a s.

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

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

___/6

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

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

- 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

$$a \setminus \left(((a \cdot a) \mid (b \cdot b))^* \right) =$$

$$b \setminus \left((b \cdot c)^* \cdot b \right) =$$

$$c \setminus \left((a \cdot b^*) \mid (b \cdot c^*) \right) =$$

$$b \setminus \left(((a \cdot b^* \cdot a) \mid b)^* \right) =$$

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 Fibonacci 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

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**).

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

___/6

- 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.

Beispiele:

`recCount FIB 2 = (1, 3)`

`recCount FIB 3 = (2, 5)`

___/12

let rec `recCount<'a, 'b> (fctnl : ( 'a -> 'b) -> ( 'a -> 'b)) : 'a -> 'b * Nat =`

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 $\text{Unit} \rightarrow \text{BA}$.

```
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 ausgewertet. 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

2. `let f() = A true`

3. `let f() = A false`

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

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

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

$$b \ \&\& \ \mathbf{false} = \mathbf{false} = \mathbf{false} \ \&\& \ b.$$

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.

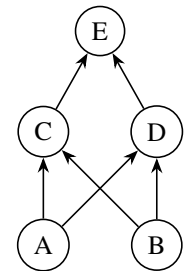
Hinweis: Definieren Sie eine (lokale) Hilfsfunktion.

___/10

let rec all (xs : List<Bool>) : Bool =

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:

- $t_1 < t_2$ 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 $\parallel$).

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		A
$C * A$	$<$	
$A * E$		$B * D$
$C \rightarrow C$	$>$	
$(D * B) \rightarrow E$	$<$	
$A \rightarrow C$		$D \rightarrow E$
$C \rightarrow D$		$A \rightarrow E$
$E \rightarrow A$	$\parallel$	
$D \rightarrow B \rightarrow E$	$<$	
$(A \rightarrow D) \rightarrow B$	$>$	

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.

```
let even0s () : Recognizer<Bool> =
```

__ / 5

- 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:

```
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 =
```

__ / 5

- c) Schreiben Sie eine Funktion `negate`: $(a : \text{Recognizer}\langle'A\rangle) \rightarrow \text{Recognizer}\langle'A\rangle$, die einen Akzeptor für die Sprache der Wörter, die *nicht* von `a` akzeptiert werden, zurückgibt.

let `negate` (`a : Recognizer<'A>`) : `Recognizer<'A>` =

___/2

- d) Schreiben Sie eine Funktion `both`: $(a : \text{Recognizer}\langle'A\rangle) \rightarrow (b : \text{Recognizer}\langle'A\rangle) \rightarrow \text{Recognizer}\langle'A\rangle$, die einen Akzeptor für die Sprache der Wörter, die von sowohl `a` als auch `b` akzeptiert werden, zurückgibt.

let `both` (`a : Recognizer<'A>`) (`b : Recognizer<'A>`) : `Recognizer<'A>` =

___/2

- e) Schreiben Sie eine Funktion `interleave`: $(a : \text{Recognizer}\langle'A\rangle) \rightarrow (b : \text{Recognizer}\langle'A\rangle) \rightarrow \text{Recognizer}\langle'A\rangle$; 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:

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

___/6

let `interleave` (`a : Recognizer<'A>`) (`b : Recognizer<'A>`) : `Recognizer<'A>` =

Fortsetzung von Aufgabe _____

Fortsetzung von Aufgabe _____