Prof. Ralf Hinze
C. Alexandru, M.Sc.
A. Dinges, M.Sc.
S. Schloßer, M.Sc.

Rheinland-Pfälzische Technische Universität Kaiserslautern Landau (RPTU)
Standort Kaiserslautern · Fachbereich Informatik · AG Programmiersprachen

Probeklausur Grundlagen der Programmierung

Dienstag, 17.12.2024

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 (14 Seiten)!
3. Sie haben 90 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:	**1a**	**1b**	**2a**	**2b**	**3**	**4ab**	**4c**	**5**	**6**
Punkte:									
Maximum:	10	10	10	10	20	10	10	20	20

Gesamtpunktzahl:	
Maximum:	120

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) $\emptyset \vdash$ **`true`** : _____

ii) $\emptyset \vdash$ _____ : `Bool`

iii) $\emptyset \vdash$ **`if true then`** _____ **`else`** _____ : Nat

iv) $\{$ `f` $\mapsto$ `Bool` $\to$ `Nat` $\} \vdash$ **`let`** `g (x: Bool): Nat = f x` : _____

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

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

$$\emptyset \vdash \texttt{let f (x: Bool): Nat = if x then 47 else 11} : \{\texttt{f} \mapsto \text{Bool} \to \text{Nat}\}$$

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!

__ /10

i) $\emptyset \vdash$ `(if` _____ `then 4 + 7 else 11) + 100` $\Downarrow 111$

ii) `fst (snd (fst ((1, (2, 3)), 4)))` $\Downarrow$ _____

iii) $\emptyset \vdash$ `let rec f (x: Nat): Bool = if x = 0 then true else f (x - 1)`

$\Downarrow$ _____

iv) $\{$`f` $\mapsto \langle \emptyset,$ `x, if` _____ `then 47 else 11`$\rangle\} \vdash$ `f false` $\Downarrow 11$

v) $\emptyset \vdash$ `fun (x: Nat)` $\rightarrow$ `fun (y: Nat)` $\rightarrow$ _____ $\Downarrow \langle \emptyset,$ `x, fun (y: Nat)` $\rightarrow$ `x + y`$\rangle$

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 `mappedInterval<'a>: (Nat -> 'a) -> Nat -> List<'a>`, die eine Funktion `f` und eine natürliche Zahl `n` nimmt und eine Liste der Form `[f n; f (n - 1); ...; f 0]` zurückgibt. Gehen Sie **strikt nach Peano Entwurfsmuster** vor.

Beispiele:

```
mappedInterval (fun n -> n + 1)     0 = [1]
mappedInterval (fun n -> n + 1)     1 = [2; 1]
mappedInterval (fun n -> n % 2 = 0) 3 = [false; true; false; true]
```

```
let rec mappedInterval<'a> (f: Nat -> 'a) (n: Nat): List<'a> =
```

__ /5

b) Schreiben Sie die Funktion `contains7: Nat -> Bool`, die prüft, ob die Ziffer 7 in der gegebenen Zahl vorkommt. Orientieren Sie sich am **Leibniz Entwurfsmuster**.

Beispiele:

```
contains7 815 = false                    contains7 4711 = true
```

```
let rec contains7 (n: Nat): Bool =
```

__ /5

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

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

Darüber hinaus verwenden wir folgende Funktionsdefinition:

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

Die Funktion `foldTree` abstrahiert das Struktur-Entwurfsmuster für Bäume, ähnlich zu `peano-pattern` aus der Vorlesung. Sie nimmt zwei Funktionen `f` und `g`. Die Funktion `g` wird im Basisfall aufgerufen, während `f` die Rückgabewerte der rekursiven Aufrufe verarbeitet.

c) Schreiben Sie eine Funktion `inorder<'a>: Tree<'a> -> List<'a>`, die die Inorder-Traversierung eines Baums zurückgibt. Verwenden Sie hierfür die Funktion `foldTree`.

Hinweis: Die Inorder-Traversierung besucht für jeden Knoten zuerst den linken und dann den rechten Teilbaum.

Beispiele:
```
inorder (Leaf 5) = [5]
inorder (Node (Leaf 5, Leaf 4)) = [5; 4]
inorder (Node (Node (Leaf 4, Leaf 7), Node (Leaf 1, Leaf 1))) = [4; 7; 1; 1]
```

```
let inorder<'a>: Tree<'a> -> List<'a> =
```

__/5

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

Beispiele:
```
mirror (Leaf 5) = Leaf 5
mirror (Node (Leaf 5, Leaf 4)) = Node (Leaf 4, Leaf 5)
mirror (Node (Node (Leaf 4, Leaf 7), Node (Leaf 1, Leaf 1))) =
        Node (Node (Leaf 1, Leaf 1), Node (Leaf 7, Leaf 4))
```

```
let mirror<'a>: Tree<'a> -> Tree<'a> =
```

__/5

Aufgabe 4 Rekursion auf Listen (__ /20 Punkte)

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

a) Wir betrachten die aus der Vorlesung bekannte Peano-Darstellung für natürliche Zahlen:

```
type Peano =
    | Zero
    | Succ of Peano
```

Schreiben Sie eine Funktion `take<'a>: Peano -> List<'a> -> List<'a>`, die eine natürliche Zahl `n` in Peano-Darstellung sowie eine Liste `xs` nimmt und die ersten `n` Elemente der Liste zurückgibt. Sollte `xs` weniger als `n` Elemente haben, dann soll die komplette Liste `xs` zurückgegeben werden.

Beispiele:

```
take (Succ Zero)               [4; 7; 1; 1] = [4]
take (Succ (Succ (Succ Zero))) ['a'; 'b']   = ['a'; 'b']
```

```
let rec take<'a> (n: Peano) (xs: List<'a>): List<'a> =
```

__ /5

b) Schreiben Sie eine Funktion `reverse<'a>: List<'a> -> List<'a>`, die die gegebene Liste spiegelt.

Beispiele:

```
reverse []         = []                    reverse [1; 2; 1]    = [1; 2; 1]
reverse ['a'; 'b'] = ['b'; 'a']            reverse [4; 7; 1; 1] = [1; 1; 7; 4]
```

```
let rec reverse<'a> (xs: List<'a>): List<'a> =
```

__ /5

c) Betrachten Sie folgende Implementierung für eine Funktion `isPalindrom: List<Nat> -> Bool`, die überprüfen soll, ob die gegebene Liste ein Palindrom ist. Eine Liste `xs` ist ein Palindrom, wenn `xs = reverse xs` ist. Hierbei ist `reverse` die Funktion aus der vorherigen Teilaufgabe, sie spiegelt die Liste.

```
1 let rec isPalindrom (xs: List<Nat>): Bool =
2     match xs with
3     | []  -> true
4     | [_] -> true
5     | head :: tail  ->
6         match reverse xs with
7         | last :: ys -> head = last && isPalindrom ys
8         | _ -> false
```

Die gegebene Implementierung ist fehlerhaft, wie folgende Beispiele zeigen:

Aufruf	**Gewünschtes Ergebnis**	**Tatsächliches Ergebnis**
`isPalindrom []`	true	true
`isPalindrom [1]`	true	true
`isPalindrom [1; 1]`	true	true
`isPalindrom [1; 2; 1]`	true	false (Fehler!)
`isPalindrom [1; 2; 3]`	false	false
`isPalindrom [1; 2; 2; 1]`	true	false (Fehler!)

Finden Sie den Fehler in der gegebenen Implementierung. Geben Sie an, in welcher Zeile der Fehler ist und wie diese Zeile richtig lauten muss. Erklären Sie in maximal 50 Wörtern, wieso Ihre Korrektur den Fehler behebt.

Es ist nicht erlaubt, die gesamte Implementierung zu ersetzen durch zum Beispiel `xs = reverse xs`. Sie dürfen nur eine einzelne Zeile korrigieren!

__ / 10

Aufgabe 5 Endliche Abbildungen (__ /20 Punkte)

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

Wir betrachten folgenden Typen, um endliche Abbildungen zu modellieren:

```
type Map<'k, 'v> = 'k -> Option<'v>
```

Ist also eine endliche Abbildung φ gegeben und `key` $\in$ dom(φ), dann gibt die zugehörige Abbildung vom Typ `Map<'k, 'v>` den Wert φ(`key`) zurück. Hierbei ist `'k` der Typ des Definitions- und `'v` der Typ des Wertebereichs. Ist `key: 'k` gegeben mit `key` $\notin$ dom(φ), gibt die zugehörige Abbildung `None` zurück.

Hinweis: Sie dürfen davon ausgehen, dass die Schlüssel vom Typ `'k` mit dem Gleichheitsoperator '=' vergleichbar sind.

a) Geben Sie die leere Abbildung $\emptyset$ an:

```
let empty<'k,'v>: Map<'k, 'v> =
```

__ /5

b) Schreiben Sie eine Funktion `insert<'k, 'v>: 'k * 'v -> Map<'k, 'v> -> Map<'k, 'v>`, die das gegebene Schlüssel-/Wertepaar in die gegebene endliche Abbildung einfügt.

```
let insert<'k,'v> (key: 'k, value: 'v) (map: Map<'k, 'v>): Map<'k, 'v> =
```

__ /5

c) Schreiben Sie eine Funktion `lookup<'k, 'v>: 'k -> Map<'k, 'v> -> Option<'v>`, die den zu dem gegebenen Schlüssel in der gegebenen endliche Abbildung zugehörigen Wert bestimmt. Wenn der Schlüssel nicht in der Abbildung enthalten ist, soll `None` zurückgegeben werden.

```
let lookup<'k, 'v> (key: 'k) (map: Map<'k, 'v>): Option<'v> =
```

__/5

d) Schreiben Sie eine Funktion `comma<'k, 'v>: Map<'k, 'v> -> Map<'k, 'v> -> Map<'k, 'v>`, die den Kommaoperator für endliche Abbildungen implementiert. Wenn ein Schlüssel in beiden endlichen Abbildungen enthalten ist, wird also der zweiten endlichen Abbildung Vorrang gegeben.

```
let comma<'k, 'v> (map1: Map<'k, 'v>) (map2: Map<'k, 'v>): Map<'k, 'v> =
```

__/5

Aufgabe 6 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.

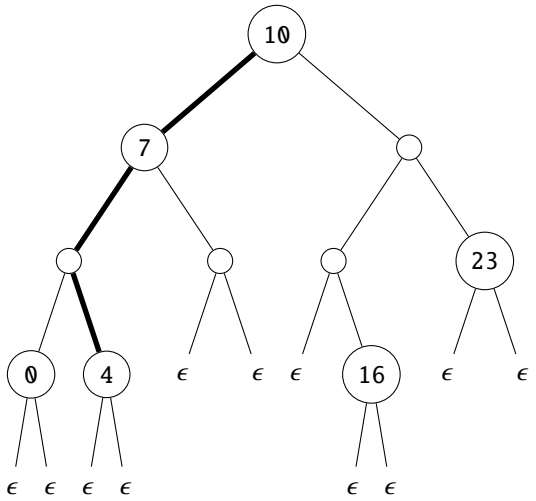
Darüber hinaus ist der folgende Typ für Pfade gegeben:

```
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> =
```

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)
```

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

Fortsetzung von Aufgabe ____