# Klausur2015/16 WS

Einführung in die Informatik 2 (IN0003) (Technische Universität München)

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

**Lehrstuhl für Sprachen und Beschreibungsstrukturen** WS 2015/16
**Einführung in die Informatik 2** **Klausur**
Prof. Dr. Helmut Seidl, Ralf Vogler, Stefan Schulze Frielinghaus

| Name | Vorname |
|---|---|
| Matrikelnummer | Unterschrift |

# Allgemeine Hinweise:

- **Bitte füllen Sie die oben angegebenen Felder vollständig aus und unterschreiben Sie!**
- **Schreiben Sie nicht mit Bleistift oder in roter/grüner Farbe!**
- **Die Arbeitszeit beträgt 90 Minuten.**
- **Prüfen Sie, ob Sie alle 14 Seiten erhalten haben.**
- **In dieser Klausur können Sie insgesamt 67 Punkte erreichen. Zum Bestehen werden maximal 25 Punkte benötigt.**
- **Bonus-Teilaufgaben sind durch (Bonus) gekennzeichnet. Insgesamt gibt es 6 Bonuspunkte.**
- **Als Hilfsmittel ist nur ein beidseitig handbeschriebenes DinA4-Blatt zugelassen.**

| **1** | **2** | **3** | **4** | **5** | **6** | $\Sigma$ | **Korrektor** |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| | | | | | | | |

**Aufgabe** [7 Punkte] **1. Multiple-Choice**

Kreuzen Sie zutreffende Antworten an bzw. geben Sie die richtige Antwort.

Punkte werden nach folgendem Schema vergeben:

- Falsche Antwort: $-\frac{1}{2}$ Punkt
- Keine Antwort: 0 Punkte
- Richtige Antwort: $\frac{1}{2}$ Punkt

Eine negative Gesamtpunktzahl wird zu 0 aufgerundet.

1. Verifikation

   (a) `false` ist die stärkste Zusicherung. ☐ Ja ☐ Nein

   (b) $x < 0$ ist eine stärkere Zusicherung als $x < -1$. ☐ Ja ☐ Nein

   (c) Zum Beweis einer Programmeigenschaft muss man an einem nachfolgenden Knoten `true` herleiten. ☐ Ja ☐ Nein

   (d) $(A \wedge B \implies C) \equiv A \implies (B \implies C)$ ☐ Ja ☐ Nein

   (e) $x \geq y$ ist eine Schleifen-Invariante für

   ```
   x = 5; y = 0; while y < x do y++; end
   ```

   ☐ Ja ☐ Nein

2. OCaml

   (a) `(f g) x` wertet sich zu dem gleichen Wert aus wie `f g x`. ☐ Ja ☐ Nein

   (b) Die durch

   ```
   let f x = let p a b = a + b in p x
   ```

   definierte Funktion `f` ist vom Typ `int -> int`. ☐ Ja ☐ Nein

   (c) Die folgende OCaml-Zeile ist fehlerfrei:

   ```
   match [1,2] with [x,y] -> x+y | z -> 0
   ```

   ☐ Ja ☐ Nein

   (d) Der Typ des Ausdrucks

   ```
   fold_left (fun a x -> x a)
   ```

   ist ☐

   (e) Die Auswertung des Ausdrucks

   ```
   let rec x = 1 in let x x = x in x 2
   ```

   liefert ☐

   (f) Der Typ des Ausdrucks (die Signatur von % findet sich im Anhang)

   ```
   fun x -> x % x
   ```

   ist ☐

3. Verifikation funktionaler Programme

(a) Die folgende abgeleitete Regel ist gültig:

$$\frac{\texttt{e} \Rightarrow \texttt{e}' :: \texttt{e}''}{(\texttt{match e with } [] \rightarrow \texttt{e}_1 \mid \texttt{x} :: \texttt{xs} \rightarrow \texttt{e}_2) = \texttt{e}_2[\texttt{e}'/\texttt{x}, \texttt{e}''/\texttt{xs}]}$$

☐ Ja ☐ Nein

(b) `fun x -> x 1` ist ein Wert. ☐ Ja ☐ Nein

(c) Ein MiniOCaml-Ausdruck ohne Funktionsapplikation terminiert immer. ☐ Ja ☐ Nein

**Aufgabe** [10 Punkte] **2. Weakest Precondition**

Berechnen Sie zuerst die folgenden schwächsten Vorbedingungen:

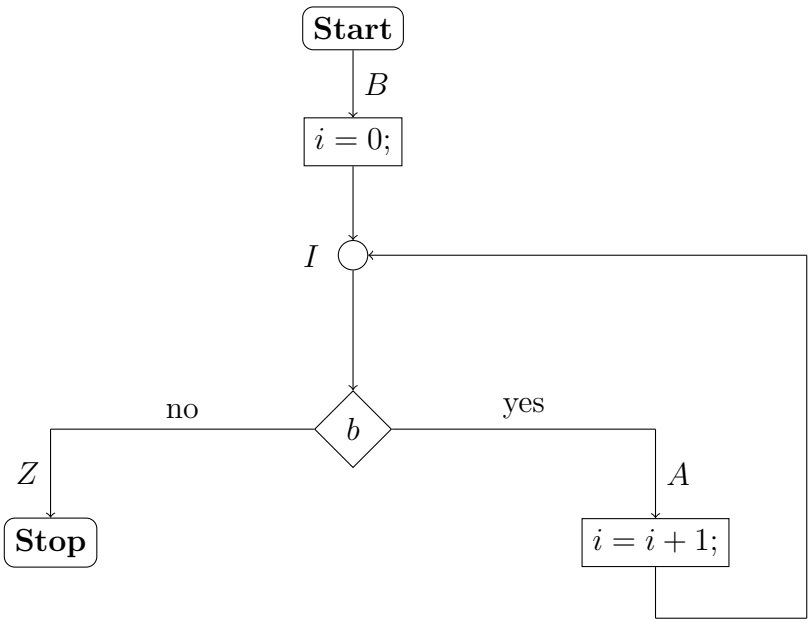
$\mathbf{WP}[\![i = i + 1;]\!](i = x \wedge x = 5) \equiv$ ____

$\mathbf{WP}[\![x < y]\!](x = 2 * y, x < 2 * y) \equiv$ ____

Gegeben sei nun das Programm

```
int x = 0;
while (b) {
  x = x+1;
}
```

mit dem Kontrollflussgraphen




Geben Sie für die folgenden Belegungen von $b$ und $Z$ die schwächsten Vorbedingungen an, welche lokal konsistent sind.

| | A | I | B |
|---|---|---|---|
| b = false<br>Z = false | | | |
| b = true<br>Z = false | | | |
| b = $i < 17$<br>Z = $i > 3$ | | | |

**Aufgabe** [9 Punkte] **3. OCaml: Sparse Vectors**

Analog zu dünn besetzten Matrizen, enthalten dünn besetzte Vektoren hauptsächlich Nullen, welche man nicht speichern will. Wir definieren uns daher einen Datentyp, der den Index (beginnend mit 0) und den Wert an dieser Position speichert.

Als Invariante soll gelten, dass nach jeder Operation nur Werte ungleich 0 in der Datenstruktur enthalten sind. Also z.B. `add [1,1] [1,-1] = []` aber auch `set 3 0 [3,1] = []`.

Implementieren Sie die folgenden Funktionen

```
type t = (int*int) list
val empty : t
val set : int -> int -> t -> t
val add : t -> t -> t
val mul : int -> t -> t
val sprod : t -> t -> int
```

wobei

**empty** Der leere Vektor

**set i v a** Setzt den Wert an Stelle $i$ des Vektors $a$ auf den Wert $v$

**add a b** Addition durch komponentenweise Summe: $\vec{a}+\vec{b} = \begin{pmatrix} a_1 \\ a_2 \\ \dots \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \dots \end{pmatrix} = \begin{pmatrix} a_1+b_1 \\ a_2+b_2 \\ \dots \end{pmatrix}$

**mul r a** Skalarmultiplikation: $r\vec{a} = \begin{pmatrix} ra_1 \\ ra_2 \\ \dots \end{pmatrix}$

**sprod a b** Standardskalarprodukt: $\langle \vec{a}, \vec{b} \rangle = \sum_{i=1}^{n} a_i b_i$

**Aufgabe** [16 Punkte] **4. OCaml: Heavy Lifting**

Wir wollen im Folgenden einen Funktor `Lift` definieren, dessen Anwendung ein Modul mit der Signatur `Base` um nützliche Funktionen erweitert.

Der Funktor soll auf beliebigen einfach polymorphen zyklenfreien Datenstrukturen arbeiten können.

Dabei ist `'a t` der Typ, `empty` liefert eine leere Datenstruktur, `insert` fügt Daten in diese ein, und `fold` faltet eine Funktion über die Daten.

Achten Sie darauf dass `fold insert empty x = x` gilt!

1. (9) Implementieren Sie den Funktor, wobei die Funktionen die vom `List`-Modul bekannte Semantik haben sollen. Wandeln Sie die Datenstruktur nicht erst in eine Liste um, sondern nutzen Sie direkt `B.fold`.

```
module type Base = sig
  type 'a t
  val empty : 'a t (* nullary/nonrec. constr. *)
  val insert : 'a -> 'a t -> 'a t (* rec. constr. *)
  val fold : ('a -> 'b -> 'b) -> 'a t -> 'b -> 'b
end

module Lift (B : Base) : sig
  include Base
  val iter : ('a -> unit) -> 'a t -> unit
  val map : ('a -> 'b) -> 'a t -> 'b t
  val filter : ('a -> bool) -> 'a t -> 'a t
  val append : 'a t -> 'a t -> 'a t
  val flatten : 'a t t -> 'a t
  val to_list : 'a t -> 'a list
  val of_list : 'a list -> 'a t
end = struct
  (* Code Aufgabe 1 *)
end
```

2. (4P) Nun wollen wir den Funktor anwenden. Geben Sie dazu ein `Base`-Modul für Listen an.

```
module List = Lift (struct
  (* Code Aufgabe 2 *)
end)
```

3. (3P) Bonus-Aufgabe: Geben Sie ein `Base`-Modul für binäre Suchbäume an.

```
module SearchTree = Lift (struct
  (* Code Bonus-Aufgabe 3 *)
end)
```

**Aufgabe** [9 Punkte] **5. OCaml: Threaded Tree**

Wir möchten nebenläufige Berechnungen auf Binärbäumen durchführen. Daten befinden sich nur in den Blättern:

```
type 'a t = Leaf of 'a | Node of 'a t * 'a t
```

Schreiben Sie eine Funktion `min` welche das minimale Element eines Baumes liefert:

```
val min : 'a t -> 'a
```

Dabei sollen innere Knoten für ihre Kinder Threads erstellen, auf das Ergebnis beider Teilbäume warten und dann ihrem Vaterknoten das Ergebnis mitteilen.

**Aufgabe** [16 Punkte] **6. MiniOCaml-Beweise: rev counter**

Gegeben sind die Definitionen

```
let rec app = fun x -> fun y -> match x
    with [] -> y
    | x::xs -> x :: app xs y
let rec rev = fun x -> match x
    with [] -> []
    | x::xs -> app (rev xs) [x]
let rec rev1 = fun x -> fun y -> match x
    with [] -> y
    | x::xs -> rev1 xs (x::y)
```

sowie

**Lemma 1** `app x [] = x`

**Lemma 2** `app (rev x) y = rev1 x y`

Beweisen Sie nun aufeinander aufbauend (das Ergebnis der jeweils vorherigen Aufgaben kann als gegeben betrachtet werden):

1. (2+4=6) `rev1 (rev1 x y) z = rev1 y (app x z)`
2. (7) `rev (rev x) = x`
3. (3) Bonus-Aufgabe: `rev (app (rev x) (rev y)) = app y x`

Geben Sie für jeden Schritt die verwendete Regel an (z.B. Def. `app`, IA, IS, Lemma 1 usw.)!

# Anhang

Funktionen die als gegeben betrachtet werden dürfen (alle anderen müssen definiert werden):

```
val ( % ) : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
val id : 'a -> 'a
val flip : ('a -> 'b -> 'c) -> 'b -> 'a -> 'c
val neg : ('a -> bool) -> 'a -> bool
val const : 'a -> 'b -> 'a
module List : sig
  val cons : 'a -> 'a list -> 'a list
  val map : ('a -> 'b) -> 'a list -> 'b list
  val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
  val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
end
```