Einführung in die Informatik 2

Prof. Dr. H. Seidl, N. Hartmann, R. Vogler 21.02.2018

Klausur

Vorname	Nachname
Matrikelnummer	Unterschrift

- Füllen Sie die oben angegebenen Felder aus.
- Schreiben Sie nur mit einem dokumentenechten Stift in schwarzer oder blauer Farbe.
- Verwenden Sie kein "Tipp-Ex" oder ähnliches.
- Die Arbeitszeit beträgt **120** Minuten.
- Prüfen Sie, ob Sie **12** Seiten erhalten haben.
- Sie können maximal **150** Punkte erreichen. Erreichen Sie mindestens **60** Punkte, bestehen Sie die Klausur.
- Als Hilfsmittel ist nur ein beidseitig beschriebenes A4-Blatt zugelassen.
- Es dürfen nur Funktionen aus den Modulen **Pervasives**, **List**, **String**, **Thread** und **Event** verwendet werden. Funktionen aus diesen Modulen dürfen ohne Angabe von Modulnamen verwendet werden.

vorzeitige Abgabe um ..... Hörsaal verlassen von ..... bis .....

1	2	3	4	5	6	7	8	Σ

.....
Erstkorrektor

.....
Zweitkorrektor

Aufgabe 1 Multiple Choice [16 Punkte]

Kreuzen Sie in den folgenden Teilaufgaben jeweils die richtigen Antworten an. Es können pro Teilaufgabe keine, einige oder alle Antworten richtig sein. Die Teilaufgaben werden isoliert bewertet, Fehler in einer Teilaufgabe wirken sich also nicht auf die erreichbare Punktzahl bei anderen Teilaufgaben aus. Don't over-think them!

1. Was gilt für ein Programm, wenn alle Zusicherungen lokal konsistent sind?

   - ☐ Die Zusicherung am Startknoten ist *true*.
   - ☒ Für alle Anweisungen s mit Vorbedingung A und Nachbedingung B gilt: $A \implies \mathsf{WP}[\![\mathtt{s}]\!](B)$.
   - ☐ Das Programm terminiert für alle Eingaben.
   - ☐ Das Programm enthält keine Schleifen.

2. Welche der nachfolgenden Aussagen gilt?

   - ☒ $x < 7 \implies x < 12 \vee y = 21$ stronger -> weaker
   - ☒ $x = 1 \wedge y = 2 \implies x > -2 \wedge y \geq x$
   - ☐ $x = y \vee x = 3 \implies y = 3$
   - ☒ $x = x - 1 \implies y > x \vee x = 21 \vee x = 2 \wedge y = 8$

3. Was ist eine hinreichende Bedingung für die Terminierung eines MiniJava Programms? sufficiency

   - ☒ Das Programm enthält keine Schleifen. loop
   - ☐ Mindestens eine Schleife im Programm wird nur endlich oft durchlaufen.
   - ☐ Im Programm kommt eine Variable r vor.
   - ☐ Am Stopknoten lässt sich *true* beweisen.

4. Was gilt für den Datentyp `type 'a t = Lama of 'a t | Dromedar` ?

   - ☒ Er definiert zwei Konstruktoren.
   - ☐ Er definiert ein Record.
   - ☒ Er ist rekursiv.
   - ☒ Er enthält eine Typ-Variable. 'a

5. Was gilt für die Funktion `let rec f a b = if b then a else f a true` ?

   - ☒ Sie ist endrekursiv.
   - ☒ Sie ist polymorph.
   - ☐ Sie terminiert nicht.
   - ☐ Sie gibt den Typ `unit` zurück.

6. Was gilt für den Ausdruck `Printf.fprintf (open_out "foo") "bar"` ?

   - [x] Er hat einen Seiteneffekt.
   - [x] Er schreibt “bar” in eine Datei “foo”.
   - [x] Er hat den Typ `unit`.
   - [ ] Er kann eine `End_of_file` Ausnahme werfen.

7. Was gilt für den Ausdruck `let g = fun x y -> 2+y x in g 1` ?

   - [x] Er enthält eine Funktion höherer Ordnung.
   - [ ] Er berechnet den Wert `3`.
   - [ ] Er enthält einen Typfehler.
   - [x] Er hat den Typ `(int -> int) -> int`.

   higher order: function using a functin as parameter

8. Für welche Voraussetzung ϕ beschreibt die Regel $\dfrac{\phi}{e_0; e_1 \Rightarrow v}$ die korrekte Big-Step Operationelle Semantik des OCaml Operators ;?

   - [ ] $\phi \equiv \; e_0 \Rightarrow v \qquad e_1$ terminiert
   - [ ] $\phi \equiv \; e_0 \Rightarrow v \qquad e_1 \Rightarrow$ `[]`
   - [ ] $\phi \equiv \; e_0 \Rightarrow v \qquad e_1 \Rightarrow 0$
   - [x] $\phi \equiv \; e_0$ terminiert $\qquad e_1 \Rightarrow v$

Aufgabe 2 OCaml-Typen [10 Punkte]

let f a = f a;;
is not a expression(Ausdruck) -1 Point

1. Geben Sie einen Ausdruck mit dem Typ `('a -> 'b) -> 'a -> 'b` .

   fun f a -> f a

2. Geben Sie einen Ausdruck mit dem Typ `('a -> 'b) -> ('c -> 'a) -> 'c -> 'b` .

   fun f g c -> f (g c) or fun f g c -> c |> g |> f

3. Welchen Typ hat der Ausdruck `fun xs -> (List.hd xs) 1` ?

   (int -> 'a) list -> 'a

4. Welchen Typ hat der Ausdruck `fun f g x y -> f (g x) (g y)` ?

   ('b -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'a -> 'c

Aufgabe 3 OCaml: Balancierte Klammern [12 Punkte]

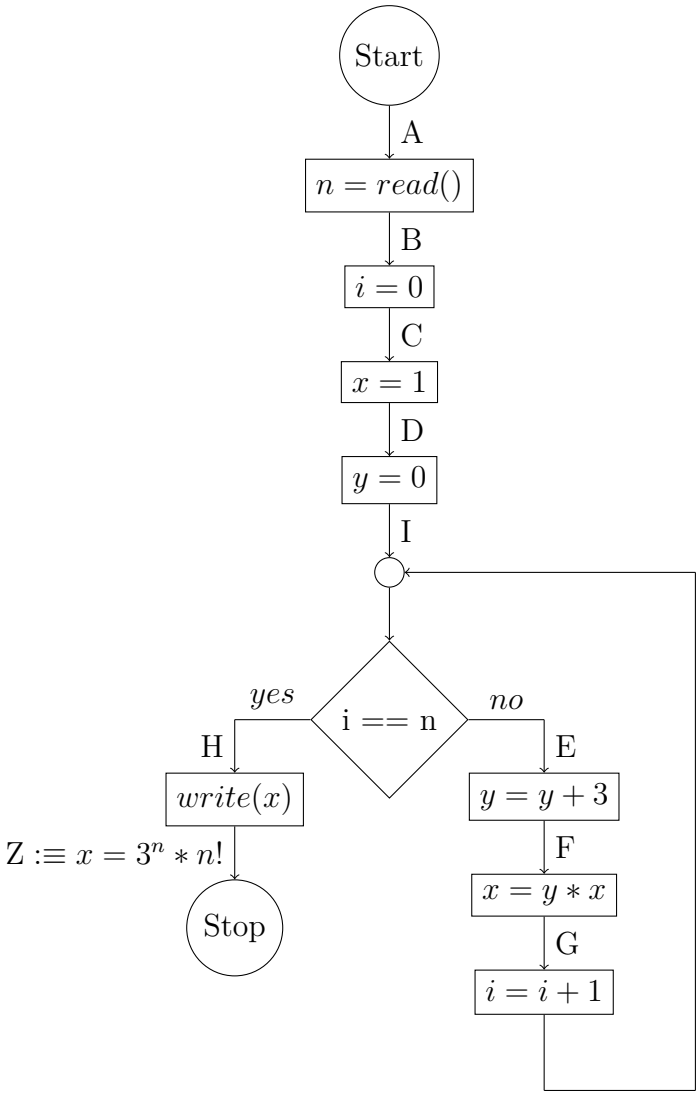
Implementieren Sie eine Funktion `is_balanced : string -> bool` welche entscheidet, ob Klammern in der Eingabe balanciert sind. Klammern sind balanciert, wenn es nie mehr schließende als öffnende und am Ende genau gleich viele öffnende wie schließende Klammern gibt.
Beispiele:

```
false = is_balanced ")("
false = is_balanced "(a)b)"
true  = is_balanced "foo"
true  = is_balanced "a(b(c)d)e"
```

Für diese Aufgabe dürfen Sie annehmen, dass es eine Funktion
`String.to_list : string -> char list` gibt, welche einen String in eine Liste seiner Zeichen umwandelt.

Aufgabe 4 Verifikation: Weakest Preconditions [22 Punkte]

Zeigen Sie mithilfe des WP-Kalküls, dass die Zusicherung Z für alle Ausführungen des folgenden Programms gilt. Terminierung muss nicht gezeigt werden.



Aufgabe 5 Big-Step Semantik, Äquivalenz [24 Punkte]

Es seien folgende Funktionsdefinitionen gegeben:

```
let rec nlen n l =
  match l with [] -> 0
  | h::t -> n + nlen n t

let rec fold_left f a l =
  match l with [] -> a
  | h::t -> fold_left f (f a h) t

let rec map f l =
  match l with [] -> []
  | h::t -> f h :: map f t

let (+) a b = a + b
```

1. Zeigen Sie mithilfe der Äquivalenzumformungen, dass für beliebige `l` und `n` gilt:

   ```
   nlen n l = fold_left (+) 0 (map (fun _ -> n) l)
   ```

2. Zeigen Sie mithilfe der Big-Step operationellen Semantik, dass der Ausdruck

   ```
   map (fun a -> a * a) [nlen 2 [3]]
   ```

   zur Liste `[4]` auswertet. Regeln der Form $v \Rightarrow v$ dürfen weggelassen werden. Dazu seien bereits folgende Teile gegeben, die Sie in Ihrem Beweis direkt verwenden dürfen:

$$\pi_{map} = \text{GD}\ \frac{\texttt{map} = \texttt{fun f l -> match l with [] -> [] | h::t -> f h::map f t}}{\texttt{map} \Rightarrow \texttt{fun f l -> match l with [] -> [] | h::t -> f h::map f t}}$$

$$\pi_{nlen} = \text{GD}\ \frac{\texttt{nlen} = \texttt{fun n l -> match l with [] -> 0 | h::t -> n+nlen n t}}{\texttt{nlen} \Rightarrow \texttt{fun n l -> match l with [] -> 0 | h::t -> n+nlen n t}}$$

Aufgabe 6 OCaml: Chain Reaction [24 Punkte]

Eine Strukturformel beschreibt eine chemische Verbindung. Dazu werden die einzelnen Bestandteile, zusammen mit einem Index, der ihre Häufigkeit angibt, aufgelistet. Kommt ein Bestandteil nur einfach vor, wird auf die Angabe eines Index verzichtet. Als Bestandteile können einzelne Atome oder andere Verbindungen vorkommen, die sich wiederum entsprechend zusammensetzen, sodass sich damit beliebige Verbindungen, wie z.B. O_2 (Sauerstoff), H_2O (Wasser), Al_2O_3 (Aluminiumoxid) oder $(NH_4)_2SO_4$ (Ammoniumsulfat), darstellen lassen. H, O, C, usw. stehen für Atome des entsprechenden Elements.

Es seien folgende OCaml-Typen zur Modellierung chemischer Verbindungen (*chemical bond*) definiert:

```
type elem = H | O | N | Al | S | (* ... *)
type bond = Atom of elem | Bond of (bond * int) list
```

Die Paare in der Liste des `Bond` Konstruktors stellen jeweils eine Teilverbindung mit ihrem Index (Häufigkeit) dar.

1. **Aufgabe:** Implementieren Sie eine Funktion `atoms : elem -> bond -> int`, die bestimmt, wie viele Atome eines Elements (1. Argument) in einer Verbindung (2. Argument) vorkommen.

Chemische Reaktionen lassen sich mithilfe einer Reaktionsgleichung darstellen, wobei die Ausgangsstoffe (Reaktanden) auf der linken und die erzeugten Verbindungen (Produkte) auf der rechten Seite des Pfeils stehen:

$$\mathrm{Al} + \mathrm{O_2} \longrightarrow \mathrm{Al_2O_3}$$

Damit eine solche Reaktion vollständig ablaufen kann, müssen allerdings die Verhältnisse der beteiligten Stoffe so angepasst werden, dass die Anzahl der Atome aller Elemente auf beiden Seiten des Reaktionspfeils dieselbe ist. Man bezeichnet die Reaktionsgleichung dann als *balanciert*. Dazu werden alle beteiligten Stoffe mit Faktoren versehen. Für obige Reaktion lautet die balancierte Reaktionsgleichung

$$4\ \mathrm{Al} + 3\ \mathrm{O_2} \longrightarrow 2\ \mathrm{Al_2O_3}$$

da auf beiden Seiten der Gleichung dann jeweils 4 Aluminium- und 6 Sauerstoffatome auftauchen.

Um eine Reaktionsgleichung darzustellen sei folgender OCaml-Typ gegeben, wobei beide Seiten der Gleichung durch eine Liste von Verbindung mit entsprechendem Faktor repräsentiert werden:

```
type reaction = { reacts : (int * bond) list; prods : (int * bond) list }
```

2. **Aufgabe:** Implementieren Sie eine Funktion `is_balanced : reaction -> bool`, die prüft, ob eine gegebene Reaktionsgleichung *balanciert* ist.

Hinweis: Die Funktion `atoms` *kann für die Implementierung von* `is_balanced` *verwendet werden, wenn Sie dies möchten.*

Aufgabe 7 OCaml (Module/Funktoren): Iterator [22 Punkte]

Ein Iterator abstrahiert das Ablaufen der Elemente einer Datenstruktur in einer bestimmten Ordnung. Der Iterator kann dabei als eine Art Zeiger oder Cursor verstanden werden, der auf ein Element der Datenstruktur verweist und nach Ausgabe dieses Elements durch entsprechende Operationen zum nächsten weiterbewegt werden kann. Dazu sei folgende Signatur definiert:

```
module type Iter = sig
  type 'a t
  type 'a s
  val init : 'a t -> 'a s
  val next : 'a s -> 'a option * 'a s
end
```

Die Funktion `init` initialisiert den Iterator so, dass dieser auf den "Anfang" der Datenstruktur verweist und beim nachfolgenden Aufruf von `next` entsprechend das erste gespeicherte Element ausgegeben wird. Von `next` wird dann das nächste Element, sowie der neue Zustand des Iterators zurückgegeben. Enthält die Datenstruktur keine (weiteren) Elemente, so gibt `next` stets `None`, sowie den unveränderten Zustand zurück. Die abstrakten Typen `t` und `s` repräsentieren den Typ der Datenstruktur, bzw. den Zustand des Iterators.

1. Implementieren Sie das Modul `ListIter`, welches obige Signatur erfüllt und über Listen beliebigen Typs iteriert.

2. Implementieren Sie das Modul `TreeIter`, welches obige Signatur erfüllt und über Bäume vom Typ `type 'a tree = Leaf | Node of 'a * 'a tree * 'a tree` iteriert. Die Werte der Knoten sollen dabei *in-order* ausgegeben werden.

3. Implementieren Sie den Funktor `ExtIter`, der einen gegebenen Iterator um die Funktionen

   ```
   next_filtered : ('a -> bool) -> 'a s -> 'a option * 'a s und
   next_mapped : ('a -> 'b) -> 'a s -> 'b option * 'a s
   ```

   erweitert. Erstere überspringt beim Iterieren alle Elemente, die das gegebene Prädikat nicht erfüllen. Letztere wendet die gegebene Funktion auf das vom Iterator gelieferte Element an.

4. Implementieren Sie den Funktor `PairIter` dessen Ergebnis die folgende Signatur erfüllt

   ```
   sig
     type ('a, 'b) t
     type ('a, 'b) s
     val init : ('a, 'b) t -> ('a, 'b) s
     val next : ('a, 'b) s -> ('a * 'b) option * ('a, 'b) s
   end
   ```

   Der Funktor bekommt zwei Module der Signatur `Iter`. Bei `next` werden jeweils Paare von Werten geliefert, wobei der erste und zweite Wert des Paars vom ersten bzw. zweiten Iterator geliefert wird. Erreicht einer der beiden gegebenen Iteratoren das Ende seiner Datenstruktur, dann liefert auch die Funktion `next` des `PairIter` den Wert `None` und den unveränderten Zustand zurück.

Aufgabe 8 OCaml (Threads): Blog-Server [20 Punkte]

Im folgenden werden Sie einen Blogserver realisieren, auf dem registrierte Benutzer ihren Blog veröffentlichen können. Wir betrachten einen *Blog* hierzu als eine Folge von veröffentlichten Texten.

```
type blog = string list
type user = string
type pass = string
type message = Post of user * pass * string
             | Read of user * blog channel
type t = message channel
```

Clients kommunizieren dann über Nachrichten mit dem Server, der durch einen `message channel` repräsentiert wird:

- Über eine `Post`-Nachricht wird ein neuer Text im Blog des Nutzers veröffentlicht. Dazu müssen der `Post`-Nachricht der Benutzername, Passwort, sowie der Text als Argumente mitgeliefert werden. Der Server muss dann zunächst überprüfen, ob der Benutzer existiert und das Passwort korrekt ist, bevor der Text an den Blog des Benutzers angefügt wird. Andernfalls wird die Nachricht einfach ignoriert.
- Mit einer `Read`-Nachricht kann der Blog eines Nutzers (1. Argument) gelesen werden. Dieser wird dazu vom Server über den mitgelieferten Antwortkanal (2. Argument) gesendet. Sollte der Benutzer nicht existieren, wird ein leerer Blog (`[]`) geschickt.

Implementieren Sie die folgenden Funktionen:

1. `start_server : (user * pass) list -> t` startet den Server (insbesondere den Server-Thread) und bekommt als Argument die assoziative Liste von Paaren aus *Namen* und *Passwort* der registrierten Nutzer übergeben. Gehen Sie davon aus, dass es bei Serverstart noch keine Blogs gibt bzw. alle Blogs leer sind.
2. `post : t -> user -> pass -> string -> unit` veröffentlicht einen Text (letztes Argument) auf dem Blog des Nutzers mit gegebenem Passwort, indem entsprechend mit dem Server kommuniziert wird.
3. `read : t -> user -> blog` fordert den Blog eines Nutzers vom Server an und gibt diesen zurück.

Beispiel:

```
let s = start_server [("userA", "passA"); ("userB", "passB")] in
post s "userB" "passB" "Welcome to my OCaml blog.";
post s "userA" "passA" "My name is A and I'm starting my own blog!";
post s "userB" "12345" "I am a hacker attacking B's blog now!";
post s "userB" "passB" "You can have threads in OCaml!";
read s "userB"
(* liefert
["Welcome to my OCaml blog."; "You can have threads in OCaml!"]
*)
```

Anhang

Big-Step Semantik

Axiome: $v \Rightarrow v$ für jeden Wert v

Tupel: $$\text{T}\ \frac{e_1 \Rightarrow v_1 \quad \ldots \quad e_k \Rightarrow v_k}{(e_1, \ldots, e_k) \Rightarrow (v_1, \ldots, v_k)}$$

Listen: $$\text{L}\ \frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 :: e_2 \Rightarrow v_1 :: v_2}$$

Globale Definitionen: $$\text{GD}\ \frac{f = e \quad e \Rightarrow v}{f \Rightarrow v}$$

Lokale Definitionen: $$\text{LD}\ \frac{e_1 \Rightarrow v_1 \quad e_0[v_1/x] \Rightarrow v_0}{\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_0 \Rightarrow v_0}$$

Funktionsaufrufe: $$\textsc{App}\ \frac{e_1 \Rightarrow \texttt{fun}\ x\ \texttt{->}\ e_0 \quad e_2 \Rightarrow v_2 \quad e_0[v_2/x] \Rightarrow v_0}{e_1\ e_2 \Rightarrow v_0}$$

Funktionsaufrufe mit mehreren Argumenten: $$\textsc{App}',\ \frac{e_0 \Rightarrow \texttt{fun}\ x_1\ \ldots\ x_k\ \texttt{->}\ e \quad e_1 \Rightarrow v_1\ \ldots\ e_k \Rightarrow v_k \quad e[v_1/x_1\ \ldots\ v_k/x_k] \Rightarrow v}{e_0\ e_1\ \ldots\ e_k \Rightarrow v}$$

Pattern Matching: $$\text{PM}\ \frac{e_0 \Rightarrow v' \equiv p_i[v_1/x_1, \ldots, v_k/x_k] \quad e_i[v_1/x_1, \ldots, v_k/x_k] \Rightarrow v}{\texttt{match}\ e_0\ \texttt{with}\ p_1\ \texttt{->}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ p_m\ \texttt{->}\ e_m \Rightarrow v}$$

— sofern v' auf keines der Muster $p_1, \ldots, p_{i-1}$ passt

Eingebaute Operatoren: $$\textsc{Op}\ \frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2 \quad v_1\ \texttt{op}\ v_2 \Rightarrow v}{e_1\ \texttt{op}\ e_2 \Rightarrow v}$$

— Unäre Operatoren werden analog behandelt.

Substitutionslemma

$$\frac{e_1 = e_2}{e[e_1/x] = e[e_2/x]}$$

OCaml Referenz

Modul List

Signatur	Erklärung
`val append : 'a list -> 'a list -> 'a list`	Concatenate two lists. Same as the infix operator `@`.
`val map : ('a -> 'b) -> 'a list -> 'b list`	`map f [a1; ...; an]` applies function `f` to `a1, ..., an`, and builds the list `[f a1; ...; f an]` with the results returned by `f`.
`val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`	`fold_left f a [b1; ...; bn]` is `f (... (f (f a b1) b2) ...) bn`.
`val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`	`fold_right f [a1; ...; an] b` is `f a1 (f a2 (... (f an b) ...))`.
`val filter : ('a -> bool) -> 'a list -> 'a list`	`filter p l` returns all the elements of the list `l` that satisfy the predicate `p`. The order of the elements in the input list is preserved.
`val exists : ('a -> bool) -> 'a list -> bool`	`exists p [a1; ...; an]` checks if at least one element of the list satisfies the predicate `p`. That is, it returns `(p a1) || (p a2) || ... || (p an)`.
`val for_all : ('a -> bool) -> 'a list -> bool`	`for_all p [a1; ...; an]` checks if all elements of the list satisfy the predicate p. That is, it returns (p a1) && (p a2) && ... && (p an).
`val assoc : 'a -> ('a * 'b) list -> 'b`	`assoc a l` returns the value associated with key `a` in the list of pairs `l`. That is, `assoc a [ ...; (a,b); ...] = b` if `(a,b)` is the leftmost binding of `a` in list `l`. Raise `Not_found` if there is no value associated with a in the list `l`.
`val assoc_opt : 'a -> ('a * 'b) list -> 'b option`	Same as `assoc`, but returns `Some b`, or `None` if there is no binding.
`val remove_assoc : 'a -> ('a * 'b) list -> ('a * 'b) list`	`remove_assoc a l` returns the list of pairs `l` without the first pair with key `a`, if any.
`val mem_assoc : 'a -> ('a * 'b) list -> bool`	Same as `assoc`, but simply return true if a binding exists, and false if no bindings exist for the given key.

Modul Thread und Event

`val create : ('a -> 'b) -> 'a -> t`	`create funct arg` creates a new thread of control, in which the function application `funct arg` is executed concurrently with the other threads of the program.
`val send : 'a channel -> 'a -> unit event`	`send c x` sends a value `x` over the channel `c`. It returns an event that occurs as soon as value is received.
`val receive : 'a channel -> 'a event`	`receive c` returns an event that occurs as soon as a value is received from the channel.
`val sync : 'a event -> 'a`	`sync e` waites for a single event `e` to occur.
`val select : 'a event list -> 'a`	`select l` waites for any event in `l` to occur. The list may contain events that already occurred.
`val new_channel : unit -> 'a channel`	`new_channel ()` creates a new channel.