TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

WS 11/12

Einführung in die Informatik 2 **Nachholklausur**
Prof. Dr. Andrey Rybalchenko, M.Sc. Ruslán Ledesma Garza 03.04.2011

Name	Vorname	Studiengang	Matrikelnummer
Hörsaal	Reihe	Sitzplatz	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 120 Minuten.**
- **Prüfen Sie, ob Sie alle 13 Seiten erhalten haben.**
- **In dieser Klausur können Sie insgesamt 120 Punkte erreichen. Zum Bestehen werden 48 Punkte benötigt.**
- **Als Hilfsmittel ist nur ein beidseitig handbeschriebenes DinA4-Blatt zugelassen. Das Merkblatt mit nützlichen Prozeduren ist als letztes Blatt an die Klausur angeheftet.**

Aufgabe 1 [30 Punkte] **Programmauswertung**

Teil 1.1 [6 Punkte]. Geben Sie den Wert der folgenden Ausdrücke an.

```
(* a *)
foldl (fn (x, s) => x @ s) [] [[1,2],[3,4]]          Wert = [3, 4, 1, 2]

(* b *)
case [1, 3] of
   x :: 3 :: z => x
 | 1 :: u      => 3                                  Wert = 1

(* c *)
let val c = ref 0 in
 map (fn x => c := !c + 1) [9,8,7]; !c
end                                                  Wert = 3
```

Teil 1.2 [7 Punkte]. Geben Sie einen vollständigen Ableitungsbaum für den Typ des Ausdrucks `f 1 + x` in der Typumgebung `[f := int -> int; x := int; + := int * int -> int]` an. Hinweis: Prozeduranwendung bindet stärker als Operatoranwendung. Siehe Anhang A für einige nützliche Typregeln.

```
(* Solution *)

T := [f := int -> int; x := int; + := int * int -> int]
```

$$\dfrac{\dfrac{\dfrac{}{T \vdash f : int \to int} \quad \dfrac{}{T \vdash 1 : int}}{T \vdash f\ 1 : int} \quad \dfrac{}{T \vdash + : int * int \to int} \quad \dfrac{}{T \vdash x : int}}{T \vdash f\ 1 + x : int}$$

Teil 1.3 [4 Punkte]. Geben Sie eine Typumgebung an, die durch die Typableitung für die folgenden Deklarationen entsteht. Die Ausgangsumgebung sei leer. Siehe Anhang A für einige nützliche Typregeln.

```
fun f (x : bool) : bool = not x
val x = f true
fun g (y : int) : int = if x then y else ~y


(* Solution *)

[f := bool -> bool; x := bool; g := int -> int]
```

Teil 1.4 [13 Punkte]. Sei $T =$ `[p := bool -> bool; q := int -> int; a := bool; + := int * int -> int]` eine Typumgebung. Geben Sie einen vollständigen Ableitungsbaum für den Typ des Ausdrucks `3 + if p a then q 2 else 0` in der Umgebung T an. Siehe Anhang A für einige nützliche Typregeln.

```
(* Solution *)

                                                                  -------------
                                          SUBTREE_A     SUBTREE_B   T |- 0 : int
-------------- --------------------------- ------------------------------------------
 T |- 3 : int   T |- + : int * int -> int      T |- if p a then q 2 else 0 : int
-------------------------------------------------------------------------------------
             T |- 3 + if p a then q 2 else 0 : int


SUBTREE_A:

---------------------- ---------------
 T |- p : bool -> bool   T |- a : bool
-------------------------------------
           T |- p a : bool

SUBTREE_B:

--------------------- ---------------
 T |- q : int -> int   T |- 2 : int
--------------------------------------
           T |- q 2 : int
```

Aufgabe 2 [30 Punkte] **Programmierung 1**

Teil 2.1 [2 Punkte].

Implikation ist eine binäre logische Verknüpfung, die den Wert `true` genau dann liefert wenn entweder das erste Argument `false` ist oder das zweite Argument `true` ist.

Geben Sie eine kartesische Prozedur `imp : bool * bool -> bool` an, die die logische Implikation implementiert.

```
(* Sample solution *)

fun imp (a, b) = not a orelse b
```

Teil 2.2 [8 Punkte].

Eine natürliche Zahl x ist ein Vielfaches einer natürlichen Zahl y, wenn es eine natürliche Zahl z gibt, so dass $x = y \cdot z$. In einem Programm können wir mithilfe des Ausdrucks `x mod y = 0` überprüfen, ob `x` ein Vielfaches von `y` ist.

Geben Sie eine endrekursive Prozedur `sum : int -> int` an, so dass `sum n` die Summe von allen Vielfachen von 2 und 3 liefert, die kleiner als `n` sind. Zum Beispiel liefert `sum 15` den Wert 18.

```
(* Sample solution *)

fun sum n =
  let
    fun sum_ s m =
      if m < n then
        if m mod 2 = 0 andalso m mod 3 = 0 then sum_ (s + m) (m + 1)
        else sum_ s (m + 1)
      else s
  in sum_ 0 1 end
```

Teil 2.3 [12 Punkte].

Geben Sie eine Prozedur `summation : (int -> int) -> int -> int -> int` an, so dass `summation f m n` den Wert $f\ m + f\ (m+1) + \ldots + f\ (n-1) + f\ n$ liefert, falls $m \leq n$ ist. Sonst gibt `summation` den Wert 0 zurück. Benutzen Sie eine while-Schleife und zwei Referenzen. Zum Beispiel liefert `summation (fn x => x+1) 1 3` den Wert 9.

```
(* Sample solution *)

fun summation m n f =
  let
    val c = ref m
    val r = ref 0
  in
    while !c <= n do
      (r := !r + f !c;
       c := !c + 1
       );
    !r
  end
```

Teil 2.4 [8 Punkte].

Geben Sie eine Prozedur `distance : int list -> int list -> int` an, so dass `distance xs ys` die Anzahl der Stellen liefert, an dennen `xs` und `ys` unterschiedliche Werte haben. Zum Beispiel liefert `distance [1,2,3] [10,2,30]` den Wert 2. Falls die Längen von `xs` und `ys` unterschiedlich sind, wird der Wert `~1` zurückgegeben.

```
(* Sample solution *)

fun distance_ acc nil nil = acc
  | distance_ acc (x :: xs) (y :: ys) = if x <> y then distance_ (acc + 1) xs ys
       else distance_ acc xs ys
  | distance_ _ _ _ = ~1

fun distance l1 l2 = distance_ 0 l1 l2
```

Aufgabe 3 [20 Punkte] **Programmierung 2**

Wir bauen eine Kalkulationstabelle. Eine Kalkulationstabelle besteht aus einer Liste von Listen von Ausdrücken.

Wie können mit Hilfe von Koordinaten auf die Ausdrücke zugreifen. Koordinaten sind Paare von natürlichen Zahlen `(x, y)`. Gegeben eine Kalkulationstabelle `zzs` benutzen wir die Zahl `x`, um das `x`-te Element von `zzs` zu bestimmen. Dann benutzen wir die Zahl `y` um innerhalb des `x`-ten Elementen auf den `y`-ten Ausdruck zuzugreifen. Das erste Listenelement befindet sich auf der ersten Position.

Jeder Ausdruck ist entweder eine Zahl, eine Positionsbeschreibung durch ein Zahlenpaar, oder eine Summe von Ausdrücken. Wie implementieren Ausdrücke wie folgt.

```
type pos = int * int
datatype exp = Const of int | Pos of pos | Sum of exp * exp
```

Teil 3.1 [4 Punkte].

Geben Sie eine Prozedur `get : 'a list list -> pos -> 'a` an, so dass `get zzs (x, y)` den Ausdruck liefert, der sich auf der Position `(x, y)` in `zzs` befindet. Falls die Position nicht gefunden werden kann wird die Ausnahme `Subscript` geworfen. Hier gibt es einige Beispielanwendungen von `get`:

- `get [[Const 1, Const 2], [Const 3, Const 4]] (1, 2)` liefert `Const 2`
- `get [[Const 1, Const 2], [Const 3, Const 4]] (2, 3)` wirft `Subscript`

```
(* Sample solution *)

fun get s (m, n) = nth (nth s m) n
```

Teil 3.2 [6 Punkte].

Geben Sie eine Prozedur `eval : exp list list -> exp -> int` an, so dass `eval zzs e` den Wert des Ausdrucks `e` in Bezug auf die Kalkulationstabelle `zzs` liefert. Dabe wird der Wert wie folgt berechnet.

a) Der Wert einer Konstante `Const c` ist die Zahl `c`.

b) Falls der Ausdruck `e` eine Positionsbeschreibung `Pos (x, y)` darstellt, dann ist der Wert von `e` gleich dem Wert des Ausdrucks auf der Position `(x, y)` in `zzs`. Falls in `zzs` kein Ausdruck auf der Position `(x, y)` vorhanden ist, wird die Ausnahme `Subscript` geworfen.

c) Der Wert einer Summe `Sum (e1, e2)` ist die Summe der Werte von `e1` und `e2` in Bezug auf `zzs`.

Die Anwendung von `eval` kann divergieren, falls es zirkuläre Abhängigkeiten zwischen Positionen gibt.

Hier gibt es einige Beispiele:

- `eval [[Const 1, Const 2], [Const 3, Const 5]] (Pos (1, 2))` liefert `2`
- `eval [[Const 1, Const 2], [Const 3, Const 5]] (Pos (1, 3))` wirft `Subscript`
- `eval [[Pos (2, 1), Const 2], [Pos (1, 1), Const 5]] (Pos (1, 1))` divergiert
- `eval [[Const 1, Const 2], [Const 3, Const 5]] (Sum (Pos (1, 2), Const 5))` liefert `7`
- `eval [[Const 1, Const 2], [Sum (Pos (1, 2), Const 5), Const 5]] (Sum (Pos (2, 1), Const 3))` liefert `10`

```
(* Sample solution *)

fun eval s (Sum (e1, e2)) = eval s e1 + eval s e2
  | eval s (Const i) = i
  | eval s (Pos pos) = eval s (get s pos)
```

Teil 3.3 [10 Punkte].

Sei `xs = [x1, ..., xN]` eine Liste von Zahlen. Geben Sie eine Prozedur `tab : int list -> exp list list * exp` an, so dass `tab xs` eine Kalkulationstabelle `zzs` und einen Ausdruck `e` liefert, die die Summe von Elementen in `xs` ausrechnen, d.h. der Wert von `eval zzs e` ist gleich dem Wert der Summe `x1+...+xN`. Falls `xs` leer ist wird die Zahl 0 als der Wert von `eval zzs e` ausgegeben.

Zum Beispiel liefert die Anwendung `tab [10, 20, 30]` die Tabelle `[[Const 10, Const 20, Const 30]]` und den Ausdruck `Sum (Sum (Pos (1, 1), Pos (1, 2)), Pos (1, 3))`.

```
(* Sample solution *)

fun tab is =
  let
    val row = map (fn i => Const i) is
    val e = case is of
      [] => Const 0
    | i :: is' =>
        #1(foldl (fn (_, (e, col)) => Sum (e, Pos (1, col)), col + 1) (Pos (1, 1), 2) is')
  in ([row], e) end
```

Aufgabe 4 [20 Punkte] **Programme als diskrete Strukturen**

Auf einer Party schüttelt jeder Gast jedem anderen Gast die Hand. Die Anzahl des Händeschüttelns zwischen n Gästen wird durch die mathematische Prozedur $h \in \mathbb{Z} \to \mathbb{Z}$ ausgerechnet.

```
h n = if n < 2 then 0 else if n = 2 then 1 else h (n - 1) + n - 1
```

Teil 4.1 [5 Punkte]. Geben Sie den Definitionsbereich und eine natürliche Terminierungsfunktion für `h` an.

- Definitionsbereich: $\mathbb{Z}$
- Terminierungsfunktion: $\lambda n \in \mathbb{Z} \,.\, max(0, n-2)$

Teil 4.2 [5 Punkte]. Geben Sie die Rekursionsfunktion und die Rekursionsrelation für `h` an.

- Rekursionsfunktion: $\lambda n \in \mathbb{Z} \,.\, \text{if } n > 2 \text{ then } \langle n-1 \rangle \text{ else } \langle\rangle$
- Rekursionsrelation: $\{(n, n') \mid n' \in \mathbb{N} \wedge n' = n-1\}$

Teil 4.3 [10 Punkte].

Wir definieren die Funktion f als $\lambda n \in \mathbb{Z}\,.\, \sum_{i=1}^{n-1} i$ und nehmen an, dass für alle $m < 0$ die Gleichheit $\sum_{i=1}^{m} i \;=\; 0$ gilt. Beweisen Sie, dass f die Ergebnisfunktion von `h` darstellt.

Sample solution. We have to check that

a) *Dom* $f \subseteq$ *Dom* p, and that

b) f satisfies the defining equations of `h`. We proceed by case distinction over n.

i) Case n < 2: Prove that $f\ n = 0$. The equation holds by the defn. of f.

ii) Case n $= 2$: Prove that $f\ n = 1$. The equation holds by the defn. of f.

iii) Case n > 2: Prove that $f\ n = f\ (n-1) + n - 1$.

$$\begin{aligned} f\ n &= \sum_{i=1}^{n-1} i \\ &= \sum_{i=1}^{n-2} i + n - 1 \\ &= f\ (n-1) + n - 1 \quad \text{by defn. of } f \end{aligned}$$

Aufgabe 5 [20 Punkte] **Programmverifikation durch induktive Beweise**

Wir betrachten eine mathematische Prozedur `exp : ℕ -> ℕ`, die wie folgt definiert wird.

$$\begin{array}{lll} exp\ 0 & = 1 & \\ exp\ n & = 2 * exp\ (n-1) & \text{falls } n > 0 \text{ und } n \text{ ungerade ist} \\ exp\ n & = exp\ (n/2) * exp\ (n/2) & \text{falls } n > 0 \text{ und } n \text{ gerade ist} \end{array}$$

Beweisen Sie, dass für jede natürliche Zahl n das Ergebnis der Anwendung $exp\ n$ gleich 2^n ist.

Sample solution. We want to prove that $exp\ n = 2^n$. We proceed by induction over n.

a) Case $n = 0$. Prove that $exp\ 0 = 2^0$. The equation holds by the definition of exp.

b) Case $n > 0$. Assume as induction hypothesis that $\forall n' < n\,.\, exp\ n' = 2^{n'}$. Prove that $exp\ n = 2^n$. We proceed by case distinction on n.

   i) Case n is odd.

$$\begin{array}{lll} exp\ n & = 2 * exp\ (n-1) & \\ & = 2 * 2^{n-1} & \text{by induction hypothesis} \\ & = 2^n & \end{array}$$

   ii) Case n is even.

$$\begin{array}{lll} exp\ n & = exp\ (n/2) * exp\ (n/2) & \\ & = 2^{n/2} * 2^{n/2} & \text{by induction hypothesis} \\ & = 2^{n/2+n/2} & \\ & = 2^n & \end{array}$$

A Anhang

$$\frac{T(b) = t}{T \vdash b : v} \qquad \frac{k \in \mathbb{Z}}{T \vdash k : \texttt{int}} \qquad \frac{T \vdash e_1 : t_1 \texttt{ -> } t_2 \quad T \vdash e_2 : t_1}{T \vdash e_1\ e_2 : t_2}$$

$$\frac{T \vdash e_1 : t_1 \quad T \vdash \circ : t_1 \texttt{ * } t_2 \texttt{ -> } t \quad T \vdash e_2 : t_2}{T \vdash e_1 \circ e_2 : t} \qquad \frac{T \vdash e_1 : \texttt{bool} \quad T \vdash e_2 : t \quad T \vdash e_3 : t}{T \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t}$$

$$\frac{T + [f := t_1 \texttt{ -> } t_2] + [b := t_1] \vdash e : t_2}{T \mid\!> \big(\text{fun } f\ (b\ :\ t_1)\ :\ t_2 = e\big) : T + [f := t_1 \texttt{ -> } t_2]} \qquad \frac{T \vdash e : t}{T \mid\!> \text{val } b = e : T + [b := t]}$$

$$\frac{T_0 \mid\!> d_1 : T_1 \quad \ldots \quad T_n \mid\!> d_n : T_{n+1}}{T_0 \mid\!> d_1\ \ldots\ d_n : T_{n+1}}$$

```
fun map f nil      = nil
  | map f (x::xr) = (f x) :: (map f xr)

map : ('a -> 'b) -> 'a list -> 'b list

fun filter f nil      = nil
  | filter f (x::xr) = if f x then x :: filter f xr
  else filter f xr

filter : ('a -> bool) -> 'a list -> 'a list

fun exists f nil      = false
  | exists f (x::xr) = f x orelse exists f xr

exists : ('a -> bool) -> 'a list -> bool

fun all f nil      = true
  | all f (x::xr) = f x andalso all f xr

all : ('a -> bool) -> 'a list -> bool

fun foldl f s nil      = s
  | foldl f s (x::xr) = foldl f (f(x,s)) xr

foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

fun foldr f s nil      = s
  | foldr f s (x::xr) = f(x, foldr f s xr)

foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

fun length nil      = 0
  | length (x::xr) = 1 + length xr

length : 'a list -> int

fun nth nil n = raise Subscript
  | nth (x :: xs) 0 = x
  | nth (x :: xs) n = nth xs (n - 1)

nth : 'a list -> int -> 'a
```