Einführung in die Informatik 2

Prof. Dr. Seidl, J. Kranz, N. Hartmann J. Brunner WS 2016/17

Übungsblatt 5 Abgabefrist: 5.12.2016

Aufgabe 5.1 (P) Grundlegende Ocamlfizierung durch fac und fib

Bearbeiten Sie folgende Teilaufgaben.

1. Implementieren Sie die Ocaml-Funktion `val fac : int -> int`, die die Fakultät einer Zahl $n \in \mathbb{N}$ berechnet. Die Fakultät ist wie folgt definiert:

$$fac(n) = \begin{cases} 1 & \texttt{für } n \leq 1 \\ n * fac(n-1) & \texttt{sonst} \end{cases}$$

2. Implementieren Sie die Ocaml-Funktion `val fib : int -> int`, welche die n-te Fibonacci-Zahl für ein $n \in \mathbb{N}_0$ berechnet. Die Fibonacci-Folge ist definiert wie folgt:

$$fib(n) = \begin{cases} n & \texttt{für } n \leq 1 \\ fib(n-1) + fib(n-2) & \texttt{sonst} \end{cases}$$

Hinweis: Testen Sie Ihre Implementierung!

Lösungsvorschlag 5.1

1.

```
let rec fac n =
  if n <= 1 then n
  else n * fac (n - 1)
```

2.

```
let rec fib n =
  if n <= 1 then n
  else fib (n - 1) + fib (n - 2)
```

Aufgabe 5.2 (P) Freude an, trotz und mit Listen

In dieser Aufgabe werden Sie Ocaml-Listen kennenlernen. Bearbeiten Sie folgende Teilaufgaben.

1. Implementieren Sie die Ocaml-Funktion

   `val swap_adjacent : int list -> int list`,

   die benachbarte Elemente einer Liste vertauscht. Einzelne Elemente am Ende der Liste werden dabei vertauschungsfrei in die Ergebnisliste übernommen.

2. Implementieren Sie die Ocaml-Funktion

```
val map2 : (int -> int -> int) -> int list -> int list,
```

die eine Funktion auf jeweils zwei Zahlen einer Liste anwenden soll. Zusätzlich soll das Ergebnis des i-ten Pärchens mit 2^i multipliziert werden. Die Ergebnisse werden als Liste zurückgegeben. Nehmen wir beispielsweise an, als Funktion f wird `fun x y -> 3*(x + y)`, als Parameterliste wird `[1; 4; 5; 0]` verwendet. Die Funktion berechnet daraus das Ergebnis $[2^0 * f(1,4); 2^1 * f(5,0)]$, welches zu `[15; 30]` evaluiert.

Hinweis: Testen Sie Ihre Implementierung!

Lösungsvorschlag 5.2

1.

```
let rec swap_adjacent = function
    a::b::tl -> b::a::(swap_adjacent tl)
  | [a] -> [a]
  | [] -> []
```

2. Direkte, weniger effiziente Implementierung:

```
let rec map2 f = function
    a::b::tl -> (f a b)::(map2 (fun x y -> 2*(f x y)) tl)
  | [_] | [] -> []
```

Effizientere Implementierung:

```
let map2 f l =
  let rec map2_inner i = function
    a::b::tl -> i*(f a b)::(map2_inner (2*i) tl)
  | [_] | [] -> [] in
  map2_inner 1 l
```

Aufgabe 5.3 (P) Ausdrucksverwertung

In dieser Aufgabe sollen einfache Typen für Ausdrücke definiert werden. Ausdrücke sollen ferner auch auswertbar sein. Bearbeiten Sie dazu folgende Teilaufgaben.

1. Definieren Sie einen Ocaml-Typen `expression` für Ausdrücke. Ein Ausdruck ist entweder eine Integer-Konstante oder binäre Operation.

2. Definieren Sie einen Ocaml-Typen `bin_operation` für binäre Operationen. Es sollen die Operationen *Addition*, *Subtraktion* und *Multiplikation* zur Verfügung stehen. Die Operationen erwarten als Parameter einen `bin_operand`-Wert (siehe unten).

3. Definieren Sie einen Ocaml-Typen `bin_operand` für binäre Operanden. Ein binärer Operand ist eine Struktur mit den Feldern *lhs* und *rhs*, die jeweils wieder Ausdrücke sind.

4. Implementieren Sie die Ocaml-Funktion `val evaluate : expression -> int`, die einen Ausdruck auswertet. Folgender Aufruf von `evaluate` evaluiert zu 7:

```
evaluate (Binop
         (Plus {lhs=(Const 1);
                rhs=(Binop
                       (Times {lhs=(Const 2);
                               rhs=(Const 3)}))
       }))
```

Hinweis: Testen Sie Ihre Implementierung!

Lösungsvorschlag 5.3

```
type expression =
    Binop of bin_operation
  | Const of int
and
  bin_operation =
    Plus of bin_operand
  | Minus of bin_operand
  | Times of bin_operand
and
  bin_operand = { lhs : expression; rhs: expression }

let rec evaluate = function
    Binop(op) -> (match op with
        Plus(operand) -> (evaluate operand.lhs) + (evaluate operand.rhs)
      | Minus(operand) -> (evaluate operand.lhs) - (evaluate
      ↪  operand.rhs)
      | Times(operand) -> (evaluate operand.lhs) * (evaluate
      ↪  operand.rhs))
  | Const(i) -> i
```

Allgemeine Hinweise zur Hausaufgabenabgabe

Die Hausaufgabenabgabe bezüglich dieses Blattes erfolgt auf zwei Wegen:

- Die Aufgabe **Moodle-Test über Ocaml** muss in Moodle bearbeitet werden.
- Alle übrigen Aufgaben werden über das Ocaml-Abgabesystem abgegeben.

Das Ocaml-Abgabesystem erreichen Sie unter `https://vmnipkow3.in.tum.de`. Sie können hier Ihre Abgaben einstellen und erhalten Feedback, welche Tests zu welchen Aufgaben erfolgreich durchlaufen. Sie können Ihre Abgabe beliebig oft testen lassen. Bitte beachten Sie, dass Sie Ihre Abgabe stets als **eine einzige UTF8-kodierte Textdatei mit dem Namen *ha5.ml* hochladen müssen**.

Aufgabe 5.4 (H) Moodle-Test über Ocaml [10 Punkte]

Zur Lösung dieser Aufgabe muss der *Ocaml-Test* in Moodle abgelegt werden. Sie können den Test mehrfach ablegen, es zählt jeweils der letzte Versuch. Es werden dabei unten aufgeführte Fragen gestellt. Vergessen Sie nicht, schließlich den *Abgabe*-Button anzuklicken.

1. Welche der folgenden Aussagen sind korrekt? ≡ steht für semantische Äquivalenz.

   ☐ `a b c ≡ a (b c)`
   ☒ `a b c ≡ (a b) c`
   ☒ `1,2::[3;4] ≡ 1, ( 2 :: [ 3; 4; ] )`
   ☒ `x ≡ ((x))`
   ☒ `[(1,2);(3,4)] ≡ [1,2;3,4]`
   ☐ `1::2::3::[] ≡ 1::[2;3]::[]`
   ☒ `1::2::3::[] ≡ 1::[2;3]`
   ☒ `x, y ≡ (x,y)`

2. Welche der folgenden Ausdrücke sind gültig (d.h. enthalten keine Syntax- oder Typfehler)?

   ☒ `[[],[[]]]`
   ☒ `[[1],[];[],[2.]]`
   ☒ `1 / 0`
   ☐ `[1.+.2.;3/4]`
   ☐ `1. / 3.`
   ☒ `[[1];[2;3];[4]]`
   ☒ `[1.+.2.,3/4]`

3. Geben Sie jeweils den Typen des Ausdrucks an. Achten Sie auf eine exakte Übereinstimmung mit der Syntax für Ocaml-Typen. Geben Sie z.B. für `fun x y -> (x (y + 1)) + 5` den Typen `(int -> int) -> int -> int` an.

   (a) `(1. +. (float 3)) :: [2.] @ [0.1]`

(b) `(fun x y z a b c -> y z x a b c) 2 (fun x y a b c -> 2 * (y + x a b c)) (fun x y z -> x + y + z)`

(c) `fun x y z t -> (x +. 1.) :: (t (y (fun a x z -> y (fun 1 2 0 -> 0) z) z))`

(d) `fun b c -> List.fold_left c 5 (0 :: b)`

(e) `fun l -> List.fold_left (fun a x -> a +. x) 3.0 (List.map (fun x -> x 42 +. 0.1) l)`

Lösungsvorschlag 5.4

1. See above.
2. See above.
3. (a) `float list`
   (b) `int -> int -> int -> int`
   (c) `float -> ((int -> int -> int -> int) -> int -> int) -> int -> (int -> float list) -> float list`
   (d) `int list -> (int -> int -> int) -> int`
   (e) `(int -> float) list -> float`

Aufgabe 5.5 (H) Aufwärmung [3 Punkte]

Bearbeiten Sie folgende Teilaufgaben.

1. Wir definieren die Superfibonacci-Funktion für $n \in \mathbb{N}_0$:

$$sf(n) = \begin{cases} n & \text{für } n \leq 2 \\ sf(n-1) + sf(n-2) + sf(n-3) & \texttt{sonst} \end{cases}$$

   Implementieren Sie die Ocaml-Funktion `val superfib : int -> int`, welche die Superfibonacci-Funktion berechnet.

2. Auf dem letzten Blatt haben wir die Cantorsche Paarungsfunktion $\pi(y, x)$ kennengelernt:

$$\pi(y, x) := x + \frac{1}{2}(x + y)(x + y + 1)$$

   Implementieren Sie die Ocaml-Funktion `val inv_cantor : int -> int * int`, die für einen gegebenen Wert $\pi(y, x)$ die Koordinaten x und y, also $\pi^{-1}(n)$, berechnet und als Tupel `(x, y)` zurückgibt.

Lösungsvorschlag 5.5

1.

```
let rec superfib n =
  if n <= 2 then n
  else (superfib (n - 1)) + (superfib (n - 2)) + (superfib (n - 3))
```

2. Langsame, aus dem Bild intuitive Version:

```
let inv_cantor n =
  let rec inv_cantor y x k =
    if k == n then (x, y) else
    if x == 0 then inv_cantor 0 (y + 1) (k + 1) else
      inv_cantor (y + 1) (x - 1) (k + 1)
  in inv_cantor 0 0 0
```

Schnellere Version, die zunächst nur in x-Richtung geht:

```
let inv_cantor n =
  (* Zunächst gehen wir nur entlang einer Achse *)
  let rec fast_upward x k =
    let dist = n - k in
    if dist > x then
      fast_upward (x + 1) (k + x + 1)
      (* Wenn wir die richtige Schiene gefunden haben, können
         wir das Ergebnis einfach berechnen *)
    else (x - dist, dist)
  in fast_upward 0 0
```

Aufgabe 5.6 (H) Lisa [3 Punkte]

In dieser Aufgabe geht es darum, die Freundschaft mit Ocaml-Listen zu vertiefen. Bearbeiten Sie dazu folgende Teilaufgaben.

1. Implementieren Sie die Ocaml-Funktion

   ```
   val scale_apply : int list -> (int -> int) list -> int list,
   ```

   die eine Liste von Funktionen und eine Liste von Integer-Zahlen erwartet. Die Funktion weist jeweils drei Integer-Zahlen a, b und c eine Funktion f aus der Liste von Funktionen zu und berechnet einen skalierten Wert x nach folgender Formel:

   $$x = a * f(b) + c$$

   Die resultierenden Werte werden als Liste zurückgegeben. Zusätzliche Funktionen oder Zahlen am Ende der Listen sollen jeweils ignoriert werden. Nehmen wir als Beispiel an, wir übergeben der Funktion die Integer-Liste `[1; 2; 3; 4; 5; 6; 7]` und

die Funktionen-Liste `[(fun x -> 2*x); (fun x -> 42)]`. Sie berechnet nun die 2-elementige Liste `[1*((fun x -> 2*x) 2) + 3; 4*((fun x -> 42) 5) + 6]`, die zu `[7; 174]` evaluiert.

2. Implementieren Sie die Ocaml-Funktion

```
val is_insert : (int -> int -> bool) -> int -> int list -> int list,
```

die eine Zahl in eine sortierte Liste an der richtigen Stelle einfügt, sodass die Sortierung beibehalten wird. Es kann dabei davon ausgegangen werden, dass die Eingabeliste sortiert ist. Die Reihenfolge der Elemente wird über die Vergleichsfunktion bestimmt, die als erster Parameter übergeben wird. Wird die Funktion `(<=)` übergeben, erwartet die Funktion eine aufsteigend sortierte Liste.

Nutzen Sie nun `is_insert`, um die Funktion

```
val insertion_sort : int list -> (int -> int -> bool) -> int list
```

zu implementieren, die eine Liste von Integer-Zahlen mit dem *InsertionSort* Sortierverfahren sortiert. Sie erwartet als zweites Argument wiederum eine Vergleichsfunktion, die derart eingesetzt werden soll, dass ein Aufruf mit `(<=)` zu einer aufsteigend sortierten Liste führt.

Lösungsvorschlag 5.6

1.

```
let scale_apply = fun vs fs ->
  let rec scale_apply acc vs = function
      f :: ftl -> (match vs with
          a :: b :: c :: vstl -> a * (f b) + c :: scale_apply acc
          ↪ vstl ftl
        | [_; _] | [_] | [] -> acc)
    | [] -> acc
  in scale_apply [] vs fs
```

2.

```
let rec is_insert f x = function
    hd :: tl -> if f x hd then
      (* Haben wir die Einfügestelle gefunden, fügen wir ein...
      ↪ *)
      x :: hd :: tl else
      (* ... sonst suchen wir weiter. *)
      hd :: (is_insert f x tl)
  | [] -> [x]

let insertion_sort l f =
  (* Innere Funktion, die jeweils den bereits sortierten Teil
```

```
    der Liste erhält *)
let rec insertion_sort sorted = function
    hd :: tl -> insertion_sort (is_insert f hd sorted) tl
  | [] -> sorted in
insertion_sort [] l
```

Aufgabe 5.7 (H) Trinette [4 Punkte]

Es seien folgende Typen gegeben, die einen binären Suchbaum für Personen repräsentieren:

```
type person = {
  name : string;
  age : int;
}
[@@deriving show]

type tree =
    Node of tree * tree * person
  | Leaf
[@@deriving show]
```

Bearbeiten Sie darauf aufbauend folgende Teilaufgaben.

1. Implementieren Sie die Ocaml-Funktion `val singleton : person -> tree`, die aus einer Person einen Baum erzeugt, der nur diese Person enthält.

2. Implementieren Sie die Ocaml-Funktion `val insert : person -> tree -> tree`, die eine Person in den gegebenen Baum einfügt, indem sie einen neuen Baum zurückliefert, der die Person an der richtigen Stelle enthält. Der Suchbaum soll nach dem Namen der Personen sortiert sein. Alphabetisch kleinere Personen sind dabei jeweils weiter links im Baum.

3. Implementieren Sie die Ocaml-Funktion

   `val to_sorted_list : tree -> person list`,

   die Personen aus dem gegebenen Baum aufsteigend sortiert in eine Liste einfügt.

Hinweis: Sie können davon ausgehen, dass alle Bäume, die als Parameter vorkommen, valide Suchbäume sind.
Hinweis: Eine Funktion zum Vergleichen von Strings findet sich im Modul `String`, welches Sie per `open String` einbinden können.
Hinweis: Durch die Annotation `[@@deriving show]` werden für die jeweiligen Typen automatisch Funktionen erzeugt, die sie in einen String umwandeln. Gegeben einen Baum `t`, kann man ihn dann per

```
print_endline ([%derive.show: tree] t);
```

auf der Konsole ausgeben.

Lösungsvorschlag 5.7

1.

```
let singleton p = Node(Leaf, Leaf, {name = p.name; age = p.age})
```

2.

```
let rec insert p = function
  (* Je nach Ordnung der Namen der einzufügenden Person und der
  ↪  Person im aktuellen Knoten, steigen wir in die richtige
  ↪  Richtung ab. *)
    Node(lhs, rhs, p_node) -> let v = compare p.name p_node.name in
    if v == -1 then Node(insert p lhs, rhs, p_node) else
    if v == 0 then Node(lhs, rhs, p_node) else
      Node(lhs, insert p rhs, p_node)
  (* An einem Blatt schließlich wird eingefügt. *)
  | Leaf -> singleton p
```

3.

```
let rec to_sorted_list = function
    Node(lhs, rhs, p_node) ->
    (to_sorted_list lhs) @ [p_node] @ (to_sorted_list rhs)
  | Leaf -> []
```