

Objective Caml version 3.06

```

(* ===== *)
(* Vorbemerkung: *)
(* *)
(* Beim Programmieren mit streams sollte man stets darauf achten, dass die *)
(* Funktion tail nicht unnötig oft aufgerufen wird, weil tail (Cons (x, f)) *)
(* einen Aufruf der Funktion f bewirkt, der beliebig aufwendig sein kann (im *)
(* Gegensatz zum Aufruf von head, der nur den Wert x zurückliefert). *)
(* ===== *)

# #use "/users/sieber/program/OCaml/Vorlesung/010605/streams.ml";;

(* ===== *)
(* Aufgabe 1: *)
(* ===== *)

(* ----- *)
(* iterate f x liefert den Strom x, f x, f (f x), ... *)
(* ----- *)

# let rec iterate f x =
  Cons (x, fun () -> iterate f (f x));;

val iterate : ('a -> 'a) -> 'a -> 'a stream = <fun>

(* ----- *)
(* Wenn s der Strom x_0, x_1, ... ist, so liefert within epsilon s die erste *)
(* Zahl x_i mit |x_{i+1} - x_i| < epsilon. Man beachte, dass tail (in der Re- *)
(* kursion) nur einmal aufgerufen wird. *)
(* ----- *)

# let rec within (epsilon: float) (s: float stream): float =
  let x = head s
  and t = tail s
  in if abs_float (x -. head t) < epsilon then x else within epsilon t;;

val within : float -> float stream -> float = <fun>

(* ----- *)
(* sqroot_stream a liefert den Strom x_0, x_1, ... mit x_0 = 1 und x_{i+1} = *)
(* (a / x_i + x_i) / 2. *)
(* ----- *)

# let sqroot_stream (a: float): float stream =
  (iterate (fun x -> (a /. x +. x) /. 2.) 1.);;

val sqroot_stream : float -> float stream = <fun>

# take 10 (sqroot_stream 2.);;

- : float list =
[1.; 1.5; 1.416666666667; 1.41421568627; 1.41421356237; 1.41421356237;
 1.41421356237; 1.41421356237; 1.41421356237; 1.41421356237]

(* ----- *)
(* In der Definition von sqroot a wählen wir epsilon in Abhängigkeit von a. *)
(* ----- *)

# let sqroot (a: float): float =
  within (a *. 1.e-15) (sqroot_stream a);;

val sqroot : float -> float = <fun>

# List.map sqroot [1.; 2.; 3.; 4.; 5.; 6.; 7.; 8.; 9.; 10.; 1.e-20];;

- : float list =
[1.; 1.41421356237; 1.73205080757; 2.; 2.2360679775; 2.44948974278;
 2.64575131106; 2.82842712475; 3.; 3.16227766017; 1e-10]

(* ----- *)
(* Selbst wenn wir epsilon sehr klein wählen (oder sogar auf Gleichheit abfra- *)
(* gen), kann sich unser Ergebnis vom Ergebnis der eingebauten Funktion sqrt *)
(* im letzten Bit unterscheiden, da irrationale Zahlen wie die Quadratwurzel *)
(* von 2 nicht exakt als floats darstellbar sind. *)
(* ----- *)

# List.map (fun x -> sqrt x -. sqroot x) [1.; 2.; 3.; 4.; 5.; 6.; 7.; 8.; 9.; 10.; 1.e-20];;

```

```

- : float list =
[0.; 2.22044604925e-16; 0.; -2.22044604925e-15; 0.; 0.; 0.; 4.4408920985e-16;
 0.; 4.4408920985e-16; 1.29246970711e-26]

(* ----- *)
(* Zur Berechnung von e^x bilden wir zunächst den Strom der Summanden y_n = *)
(* x^n / n!. Es gilt y_0 = 1 und y_{n+1} = y_n * x / (n+1). Dieser Strom lässt *)
(* sich nicht mit iterate erzeugen, weil y_{n+1} nicht nur von y_n, sondern *)
(* auch von n abhängt. Deshalb bilden wir zuerst den Strom der Paare (n, y_n) *)
(* und projizieren dann auf die zweite Komponente. *)
(* ----- *)

# let pairs x = iterate (fun (n, y) -> (n + 1, y *. x /. float (n + 1))) (0, 1.);;

val pairs : float -> (int * float) stream = <fun>

# let ys x = map snd (pairs x);;

val ys : float -> float stream = <fun>

# take 20 (ys 1.);;

- : float list =
[1.; 1.; 0.5; 0.1666666666667; 0.0416666666667; 0.0083333333333333;
 0.001388888888889; 0.000198412698413; 2.48015873016e-05; 2.7557319224e-06;
 2.7557319224e-07; 2.50521083854e-08; 2.08767569879e-09; 1.60590438368e-10;
 1.14707455977e-11; 7.64716373182e-13; 4.77947733239e-14; 2.81145725435e-15;
 1.56192069686e-16; 8.22063524662e-18]

(* ----- *)
(* Die unendliche Summe der y_n können wir approximieren, indem wir die Summa- *)
(* tion abbrechen, sobald der Absolutbetrag des aktuellen Summanden <= epsilon *)
(* ist. *)
(* ----- *)

# let rec sum_within epsilon s =
  let x = head s
  in if abs_float x <= epsilon then x else x +. sum_within epsilon (tail s);;

val sum_within : float -> float stream -> float = <fun>

(* ----- *)
(* Zur Berechnung von e^x sollte man epsilon wieder von x abhängig machen. *)
(* Die Wahl eines passenden epsilon ist allerdings schwierig. *)
(* ----- *)

# let e x = sum_within (abs_float x *. 1.e-15) (ys x);;

val e : float -> float = <fun>

# List.map e [0.; 0.1; 1.; -1.; 10.; -10.; 1.e-10];;

- : float list =
[1.; 1.10517091808; 2.71828182846; 0.367879441171; 22026.4657948;
 4.53999295758e-05; 1.0000000001]

# List.map (fun x -> exp x -. e x) [0.; 0.1; 1.; -1.; 10.; -10.; 1.e-10];;

- : float list = [0.; 0.; 0.; 0.; 3.63797880709e-12; 1.86733624169e-13; 0.]

(* ===== *)
(* Aufgabe 2: *)
(* ===== *)

# let rec add (s1: int stream) (s2: int stream): int stream =
  Cons (head s1 + head s2, fun () -> add (tail s1) (tail s2));;

val add : int stream -> int stream -> int stream = <fun>

(* ----- *)
(* Wenn man powers_2 zu sich selbst addiert, erhält man tail powers_2. Daraus *)
(* erhält man wieder powers_2, indem man 1 vorne anhängt. *)
(* ----- *)

# let rec powers_2 = Cons (1, fun () -> add powers_2 powers_2);;

val powers_2 : int stream = Cons (1, <fun>)

```

```

# take 15 powers_2;;

- : int list =
[1; 2; 4; 8; 16; 32; 64; 128; 256; 512; 1024; 2048; 4096; 8192; 16384]

(* ----- *)
(* Sei fibs der Strom der Fibonaccizahlen. Indem man fibs und tail fibs ad- *)
(* diert, erhält man tail (tail fibs). Daraus erhält man wieder fibs, indem man *)
(* 0 und 1 vorne anhängt. *)
(* ----- *)

# let rec fibs = Cons (0, fun () -> Cons (1, fun () -> add fibs (tail fibs)));;

val fibs : int stream = Cons (0, <fun>)

# take 20 fibs;;

- : int list =
[0; 1; 1; 2; 3; 5; 8; 13; 21; 34; 55; 89; 144; 233; 377; 610; 987; 1597;
 2584; 4181]

(* ----- *)
(* Das Problem, die Zahlen der Form  $2^i * 3^j * 5^k$  (mit  $i, j, k \geq 0$ ) in auf- *)
(* steigender Reihenfolge zu erzeugen, stammt von Richard Hamming. Sei H die *)
(* Menge dieser Zahlen. Dann ist jede Zahl aus H (außer der kleinsten Zahl 1) *)
(* das 2-, 3- oder 5-fache einer kleineren Zahl aus H. Also ist H die Vereini- *)
(* gung der Mengen  $\{1\}$ ,  $2 * H$ ,  $3 * H$  und  $5 * H$ , wobei  $a * H$  definiert ist als *)
(* die Menge  $\{a * x \mid x \in H\}$ . Entsprechendes gilt für den Strom aller Zahlen *)
(* aus H: An die Stelle der Vereinigung tritt hier die Operation merge, zur *)
(* "Skalarmultiplikation" eines Stroms s mit einer Zahl a benutzen wir die *)
(* Funktion scale, und das kleinste Element 1 wird mit Cons hinzugefügt. *)
(* ----- *)

# let rec merge (s1: int stream) (s2: int stream): int stream =
  let x1 = head s1
  and x2 = head s2
  in
  if x1 < x2
  then Cons (x1, fun () -> merge (tail s1) s2)
  else
    if x1 = x2
    then Cons (x1, fun () -> merge (tail s1) (tail s2))
    else Cons (x2, fun () -> merge s1 (tail s2));;

val merge : int stream -> int stream -> int stream = <fun>

# let scale (a: int) = map (fun x -> a * x);;

val scale : int -> int stream -> int stream = <fun>

# let rec hamming =
  Cons (1, fun () -> merge (scale 2 hamming)
    (merge (scale 3 hamming) (scale 5 hamming)));;

val hamming : int stream = Cons (1, <fun>)

# take 100 hamming;;

- : int list =
[1; 2; 3; 4; 5; 6; 8; 9; 10; 12; 15; 16; 18; 20; 24; 25; 27; 30; 32; 36; 40;
 45; 48; 50; 54; 60; 64; 72; 75; 80; 81; 90; 96; 100; 108; 120; 125; 128;
 135; 144; 150; 160; 162; 180; 192; 200; 216; 225; 240; 243; 250; 256; 270;
 288; 300; 320; 324; 360; 375; 384; 400; 405; 432; 450; 480; 486; 500; 512;
 540; 576; 600; 625; 640; 648; 675; 720; 729; 750; 768; 800; 810; 864; 900;
 960; 972; 1000; 1024; 1080; 1125; 1152; 1200; 1215; 1250; 1280; 1296; 1350;
 1440; 1458; 1500; 1536]

(* ===== *)
(* Aufgabe 3: *)
(* ===== *)

(* ----- *)
(* Bei der folgenden Definition von streams wird nicht nur die Berechnung von *)
(* tail s, sondern auch die von head s verzögert, d.h. diese streams sind - im *)
(* Gegensatz zu den in der Vorlesung definierten - "fully lazy". *)
(* ----- *)

```

```

# type 'a stream =
  Nil
  | Cons of (unit -> 'a * 'a stream);;

type 'a stream = Nil | Cons of (unit -> 'a * 'a stream)

(* ----- *)
(* Auf diesen streams müssen wir natürlich alle Funktionen neu definieren. *)
(* ----- *)

# let head s =
  match s with
  | Cons f -> fst (f ())
  | _ -> raise Empty_stream;;

val head : 'a stream -> 'a = <fun>

# let tail s =
  match s with
  | Cons f -> snd (f ())
  | _ -> raise Empty_stream;;

val tail : 'a stream -> 'a stream = <fun>

# let rec take n s =
  match (n, s) with
  | (0, _) | (_, Nil) -> []
  | _ -> head s :: take (n - 1) (tail s);;

val take : int -> 'a stream -> 'a list = <fun>

# let rec filter p s =
  let x = head s
  in
  if p x
  then Cons (fun () -> (x, filter p (tail s)))
  else filter p (tail s);;

val filter : ('a -> bool) -> 'a stream -> 'a stream = <fun>

# let rec sieve_by m = filter (fun x -> x mod m <> 0);;

val sieve_by : int -> int stream -> int stream = <fun>

# let rec sieve_by_all s =
  let m = head s
  in Cons (fun () -> (m, sieve_by_all (sieve_by m (tail s))));;

val sieve_by_all : int stream -> int stream = <fun>

# let rec ints_from m =
  Cons (fun () -> (m, ints_from (m + 1)));;

val ints_from : int -> int stream = <fun>

# let primes = sieve_by_all (ints_from 2);;

val primes : int stream = Cons <fun>

# take 15 primes;;

- : int list = [2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41; 43; 47]

(* ----- *)
(* Diese Lösung erweist sich als sehr ineffizient. Das liegt daran, dass head *)
(* und tail getrennt voneinander die Funktion f () aufrufen. Also ergibt sich *)
(* exponentielle Laufzeit, sobald man head und tail beide in einer Rekursion *)
(* aufruft. Man sollte also nicht mit head und tail arbeiten, sondern besser *)
(* mit einer Funktion force, die nur die Verzögerung aufhebt. Das Ergebnis von *)
(* force ist dann ein Paar, auf dessen Komponenten man mit den Projektionen *)
(* oder mit pattern matching zugreifen kann. *)
(* ----- *)

# let force (s: 'a stream): 'a * 'a stream =
  match s with
  | Nil -> raise Empty_stream
  | Cons f -> f ();;

```

```

val force : 'a stream -> 'a * 'a stream = <fun>

# let rec take n (s: 'a stream) =
  match (n, s) with
  | (0, _) | (_, Nil) -> []
  | _ -> let (x, t) = force s in x :: take (n - 1) t;;

val take : int -> 'a stream -> 'a list = <fun>

# let rec filter p s =
  let (x, t) = force s
  in
  if p x
  then Cons (fun () -> (x, filter p t))
  else filter p t;;

val filter : ('a -> bool) -> 'a stream -> 'a stream = <fun>

# let rec sieve_by m = filter (fun x -> x mod m <> 0);;

val sieve_by : int -> int stream -> int stream = <fun>

# let rec sieve_by_all s =
  let (x, t) = force s
  in Cons (fun () -> (x, sieve_by_all (sieve_by x t)));;

val sieve_by_all : int stream -> int stream = <fun>

# let rec ints_from m =
  Cons (fun () -> (m, ints_from (m + 1)));;

val ints_from : int -> int stream = <fun>

# let primes = sieve_by_all (ints_from 2);;

val primes : int stream = Cons <fun>

(* ----- *)
(* Jetzt geht's effizienter. *)
(* ----- *)

# take 300 primes;;

- : int list =
[2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41; 43; 47; 53; 59; 61; 67; 71;
 73; 79; 83; 89; 97; 101; 103; 107; 109; 113; 127; 131; 137; 139; 149; 151;
 157; 163; 167; 173; 179; 181; 191; 193; 197; 199; 211; 223; 227; 229; 233;
 239; 241; 251; 257; 263; 269; 271; 277; 281; 283; 293; 307; 311; 313; 317;
 331; 337; 347; 349; 353; 359; 367; 373; 379; 383; 389; 397; 401; 409; 419;
 421; 431; 433; 439; 443; 449; 457; 461; 463; 467; 479; 487; 491; 499; 503;
 509; 521; 523; 541; 547; 557; 563; 569; 571; 577; 587; 593; 599; 601; 607;
 613; 617; 619; 631; 641; 643; 647; 653; 659; 661; 673; 677; 683; 691; 701;
 709; 719; 727; 733; 739; 743; 751; 757; 761; 769; 773; 787; 797; 809; 811;
 821; 823; 827; 829; 839; 853; 857; 859; 863; 877; 881; 883; 887; 907; 911;
 919; 929; 937; 941; 947; 953; 967; 971; 977; 983; 991; 997; 1009; 1013;
 1019; 1021; 1031; 1033; 1039; 1049; 1051; 1061; 1063; 1069; 1087; 1091;
 1093; 1097; 1103; 1109; 1117; 1123; 1129; 1151; 1153; 1163; 1171; 1181;
 1187; 1193; 1201; 1213; 1217; 1223; 1229; 1231; 1237; 1249; 1259; 1277;
 1279; 1283; 1289; 1291; 1297; 1301; 1303; 1307; 1319; 1321; 1327; 1361;
 1367; 1373; 1381; 1399; 1409; 1423; 1427; 1429; 1433; 1439; 1447; 1451;
 1453; 1459; 1471; 1481; 1483; 1487; 1489; 1493; 1499; 1511; 1523; 1531;
 1543; 1549; 1553; 1559; 1567; 1571; 1579; 1583; 1597; 1601; 1607; 1609;
 1613; 1619; 1621; 1627; 1637; 1657; 1663; 1667; 1669; 1693; 1697; 1699;
 1709; 1721; 1723; 1733; 1741; 1747; 1753; 1759; 1777; 1783; 1787; 1789;
 1801; 1811; 1823; 1831; 1847; 1861; 1867; 1871; 1873; 1877; 1879; 1889;
 1901; 1907; 1913; 1931; 1933; 1949; 1951; 1973; 1979; ...]

#

```