Zadanie G: SML - Strumienie

Strumień to nieskończona lista obliczana leniwie (i spamiętywana). W każdym momencie wyewaluowany jest pewien prefiks strumienia - pozostała część jest reprezentowana jako funkcja zeroargumentowa. Strumienie można definiować rekursywnie, wykorzystując w definicji ten sam strumień (albo nawet wiele współzależnych strumieni na raz). Potrzebne definicje są poniżej:

```
local
  datatype 'a memo_cell = Value of 'a | Computation of unit -> 'a
  exception CellInvalid
  fun make_memo () =
  let
    val cell = ref (Computation (fn _ => raise CellInvalid))
    fun res () =
        case !cell of
          Computation h => (
            let val r = h () in
              cell := Value r;
              r
            end)
        | Value y => y
  in
    (cell, res)
  end
in
  (* fn : ((unit -> 'a) -> 'b * (unit -> 'a)) -> 'b
   * smemo pozwala spamiętywać obliczenia oraz pełni rolę operatora
   * punktu stałego (ze względu na brak rekursji na wartościach
   * w SMLu).
   * Zadaniem 'f' jest skonstruować obliczenie do spamiętania
   * oraz dowolną wartość, którą zwraca memo.
   * Argumentem funkcji 'f' jest to samo obliczenie, które
   * 'f' skonstruuje - ale w opakowanej wersji ze spamiętaniem.
   *)
  fun memo f =
  let
    val (mcell, memoized_computation) = make_memo ()
    val (result, computation) = f memoized_computation
  in
    mcell := Computation computation;
    result
  end
end

(* Strumień to funkcja zeroargumentowa zwracająca
 * głowę strumienia oraz ogon. Ze względu na rekursywny
 * typ, funkcja opakowana jest w konstruktor Stream
 *)
datatype 'a stream = Stream of unit -> 'a * 'a stream
fun seval (Stream f) = f ();
fun shd s = #1 (seval s);
fun stl s = #2 (seval s);
```

```
(* fn : ('a stream -> 'a * 'a stream) -> 'a stream
 * smemo tworzy spamiętany strumień. Ten strumień jest
 * przekazywany do funkcji 'f' jako argument oraz
 * zwracany przez smemo. Funkcja 'f' konstruuje
 * początek strumienia - głowę oraz ogon.
 *)
fun smemo f = memo (fn the_stream =>
  let val wrapped_stream = Stream the_stream
      in (wrapped_stream, fn () => f wrapped_stream)
  end
)
```

Zadanie polega na implementacji szeregu funkcji operujących na strumieniach. Strumienie powinny być spamiętane używając funkcji smemo (ewentualnie memo). Nie można korzystać z "niefunkcyjnych" elementów SMLa, takich jak referencje, array'e, vector'y.

W pliku .sml powinny znaleźć się definicje (proszę nie umieszczać definicji smemo, memo, stream):

```
(* Strumień stały (x, x, x, ...) w pamięci O(1) *)
val sconst = fn : 'a -> 'a stream
(* N-ty element strumienia, numerowanie od 0 *)
val snth = fn : int -> 'a stream -> 'a
(* Lista pierwszych N elementów strumienia *)
val stake = fn : int -> 'a stream -> 'a list

(* 'smap f s' to strumień zmapowany przez f:
    [f s_0, f s_1, f s_2, ...] *)
val smap = fn : ('a -> 'b) -> 'a stream -> 'b stream
(* 'smap1 f s' to strumień ze zmapowanym pierwszym elementem:
    [f s_0, s_1, s_2, ...] *)
val smap1 = fn : ('a -> 'a) -> 'a stream -> 'a stream

(* 'snat s z' to strumień [z, s z, s (s z), s (s (s z)), ...] *)
val snat = fn : ('a -> 'a) -> 'a -> 'a stream

(* 'stab f' to strumień [f 0, f 1, f 2, ...] *)
val stab = fn : (int -> 'a) -> 'a stream

(* 'szip s t' to strumień [(s_0, t_0), (s_1, t_1), ...] *)
val szip = fn : 'a stream -> 'b stream -> ('a * 'b) stream
(* 'szipwith f s t' to strumień [f (s_0, t_0), f (s_1, t_1), ...] *)
val szipwith = fn : ('a * 'b -> 'c) -> 'a stream -> 'b stream -> 'c stream

(* 'sfoldl f start s' to strumień:
  [start, f(start, s_0), f(f(start, s_0), s_1), f(f(f(start, s_0), s_1), s_2), ...] *)
val sfoldl = fn : ('a * 'b -> 'a) -> 'a -> 'b stream -> 'a stream
(* 'srev s' to strumień:
  [[s_0], [s_1, s_0], [s_2, s_1, s_0], ...] *)
val srev = fn : 'a stream -> 'a list stream

(* 'sfilter p s' to strumień s po usunięciu elementów,
```

```
   dla których predykat p jest fałszywy. Jeśli w strumieniu
   s jest skończenie wiele (n) elementów spełniających p,
   to próba odczytania n+1-szego elementu może prowadzić
   do zapętlenia. Proszę pamiętać, że sfilter ma być
   leniwy - tzn. samo wywołanie sfilter nie może skanować
   wejściowego strumienia w poszukiwaniu głowy spełniającej
   predykat. To ma nastąpić dopiero w trakcie czytania zwróconego
   strumienia. *)
val sfilter = fn : ('a -> bool) -> 'a stream -> 'a stream
(* 'stakewhile p s' to lista początkowych elementów strumienia,
   dla których p jest prawdziwy. Jeśli p jest prawdziwy
   dla nieskończenie wielu elementów, to funkcja może nie skończyć
   działania *)
val stakewhile = fn : ('a -> bool) -> 'a stream -> 'a list

(* srepeat tworzy strumień elementów powtarzających się cyklicznie.
   Zakładamy, że podana lista jest niepusta. Odczytanie pierwszych
   k elementów ze strumienia zajmuje czas O(k). Pamięć ograniczona
   jest przez O(n) gdzie n to liczba elementów w danej liście. Programy
   wykorzystujące więcej pamięci mogą dostać RTE lub TLE. *)
val srepeat = fn : 'a list -> 'a stream
(* spairs s to strumień:
   [(s_0, s_1), (s_2, s_3), (s_4, s_5), ...] *)
val spairs = fn : 'a stream -> ('a * 'a) stream

(* dzieli strumień s na n strumieni:
   [[s_0, s_n, s_{2n}, s_{3n} ...]
    [s_1, s_{n+1}, s_{2n+1}, ...]
    ...
    [s_(n-1), s_{2n-1}, s_{3n-1}, ...]]
   UWAGA:
    Jeżeli z otrzymanych strumieni odczytujemy łącznie k elementów,
    a największy numer odczytanego elementu w oryginalnym strumieniu
    to l, to czas działania jest ograniczony przez O(n+k+l). Tzn.,
    zwrócone strumienie dzielą pracę i nie przeskakują niezależnie elementów
    w czasie O(nk). *)
val ssplitn = fn : int -> 'a stream -> 'a stream list

(* Operacja odwrotna do ssplitn. Przeplata dane strumienie tworząc
   wspólny strumień *)
val sinterleave = fn : 'a stream list -> 'a stream
```

Uwaga1: Funkcje typu sfoldl, które są sparametryzowane funkcjami, nie powinny wielokrotnie wywoływać tych funkcji bez potrzeby (na tych samych argumentach). Pamiętajmy, że sml nie spamiętuje wielokrotnych wywołań.

Uwaga2: Należy czytać dokładnie tyle wartości ze strumieni wejściowych ile potrzeba do wyliczenia żądanych wartości w strumieniach wyjściowych. Wynika to stąd, że wyliczenie niektórych wartości w strumieniach wejściowych może być znacznie droższe niż innych. Programy, które odwołują się do większej części strumieni niż trzeba mogą dostać TLE.

Przykład

```
- val nat = snat (fn x => x + 1) 0;
val nat = Stream fn : int stream
- shd nat;
val it = 0 : int
- stl nat;
val it = Stream fn : int stream
- shd (stl nat);
val it = 1 : int
- snth 10 nat;
val it = 10 : int
- stake 7 nat;
val it = [0,1,2,3,4,5,6] : int list
- val twos = sconst 2;
val twos = Stream fn : int stream
- stake 3 twos;
val it = [2,2,2] : int list
- stake 10 (smap1 (fn x => x + 7) nat);
val it = [7,1,2,3,4,5,6,7,8,9] : int list
- stake 10 (smap (fn x => x * x) nat);
val it = [0,1,4,9,16,25,36,49,64,81] : int list
- stake 10 (stab (fn x => (x, x*x, x*x*x)));
val it =
  [(0,0,0),(1,1,1),(2,4,8),(3,9,27),(4,16,64),(5,25,125),(6,36,216),
   (7,49,343),(8,64,512),(9,81,729)] : (int * int * int) list
- stake 10 (szip nat (szipwith (fn (x, y) => [x+y, x*y]) twos nat));
val it =
  [(0,[2,0]),(1,[3,2]),(2,[4,4]),(3,[5,6]),(4,[6,8]),(5,[7,10]),(6,[8,12]),
   (7,[9,14]),(8,[10,16]),(9,[11,18])] : (int * int list) list
- stake 8 (sfoldl (fn (x, y) => x * (y+1)) 1 nat);
val it = [1,1,2,6,24,120,720,5040] : int list
- stake 5 (srev nat);
val it = [[0],[1,0],[2,1,0],[3,2,1,0],[4,3,2,1,0]] : int list list
- stake 10 (sfilter (fn x => x mod 3 <> 0) nat);
val it = [1,2,4,5,7,8,10,11,13,14] : int list
- stakewhile (fn x => x <> 7) nat;
val it = [0,1,2,3,4,5,6] : int list
- stake 15 (srepeat [1,2,3]);
val it = [1,2,3,1,2,3,1,2,3,1,2,3,...] : int list
- stake 10 (spairs nat);
val it =
  [(0,1),(2,3),(4,5),(6,7),(8,9),(10,11),(12,13),(14,15),(16,17),(18,19)]
  : (int * int) list
- foldr (fn (str, lst) => (stake 7 str)::lst) [] (ssplitn 5 nat);
val it =
  [[0,5,10,15,20,25,30],[1,6,11,16,21,26,31],[2,7,12,17,22,27,32],
   [3,8,13,18,23,28,33],[4,9,14,19,24,29,34]] : int list list
- stake 20 (sinterleave (ssplitn 5 nat));
val it = [0,1,2,3,4,5,6,7,8,9,10,11,...] : int list
```