```
(* Author: Flavio Lerda *)
(* 15-212 Section C     *)
(* Code from recitation *)
(* March, 24th 2004     *)

exception Unimplemented

(* Taken from the lecture code *)
(* BASIC_STREAM defines the visible "core" of streams *)
signature BASIC_STREAM =
  sig
    type 'a stream
    datatype 'a front = Empty | Cons of 'a * 'a stream

    (* Lazy stream construction and exposure *)
    val delay : (unit → 'a front) → 'a stream
    val expose : 'a stream → 'a front

    (* Eager stream construction *)
    val empty : 'a stream
    val cons : 'a * 'a stream → 'a stream
  end;

structure BasicStream :> BASIC_STREAM =
  struct
    datatype 'a stream = Stream of unit → 'a front
    and 'a front = Empty | Cons of 'a * 'a stream

    fun delay (d) = Stream(d)
    fun expose (Stream(d)) = d ()

    val empty = Stream (fn () ⇒ Empty)
    fun cons (x, s) = Stream (fn () ⇒ Cons (x, s))
  end;

(* STREAM extends BASIC_STREAM by operations *)
(* definable without reference to the implementation *)
signature STREAM =
  sig
    include BASIC_STREAM

    exception EmptyStream
    val null : 'a stream → bool
    val hd : 'a stream → 'a
    val tl : 'a stream → 'a stream

    val map : ('a → 'b) → 'a stream → 'b stream
    val filter : ('a → bool) → 'a stream → 'a stream
    val exists : ('a → bool) → 'a stream → bool

    val take : 'a stream * int → 'a list
    val drop : 'a stream * int → 'a stream

    val tabulate : (int → 'a) → 'a stream

    val append : 'a stream * 'a stream → 'a stream
  end;

functor Stream
```

```
  (structure BasicStream : BASIC_STREAM) :> STREAM =
struct
  open BasicStream

  exception EmptyStream

  (* functions null, hd, tl, map, filter, exists, take, drop *)
  (* parallel the functions in the List structure *)
  fun null (s) = null' (expose s)
  and null' (Empty) = true
    | null' (Cons _) = false

  fun hd (s) = hd' (expose s)
  and hd' (Empty) = raise EmptyStream
    | hd' (Cons (x,s)) = x

  fun tl (s) = tl' (expose s)
  and tl' (Empty) = raise EmptyStream
    | tl' (Cons (x,s)) = s

  (* too eager---doesn't terminate on infinite streams
   fun map f s = map' f (expose s)
   and map' f (Empty) = empty
   | map' f (Cons(x,s)) = cons (f(x), map f s)
   *)

  fun map f s = delay (fn () => map' f (expose s))
  and map' f (Empty) = Empty
    | map' f (Cons(x,s)) = Cons (f(x), map f s)

  fun filter p s = delay (fn () => filter' p (expose s))
  and filter' p (Empty) = Empty
    | filter' p (Cons(x,s)) =
      if p(x) then
        Cons (x, filter p s)
      else
        filter' p (expose s)

  fun exists p s = exists' p (expose s)
  and exists' p (Empty) = false
    | exists' p (Cons(x,s)) =
      p(x) orelse exists p s

  (* take (s,n) converts the first n elements of n to a list *)
  (* raises Subscript if n < 0 or n >= length(s) *)
  fun takePos (s, 0) = nil
    | takePos (s, n) = take' (expose s, n)
  and take' (Empty, _) = raise Subscript
    | take' (Cons(x,s), n) = x::takePos(s, n-1)

  fun take (s,n) = if n < 0 then raise Subscript else takePos (s,n)

  fun dropPos (s, 0) = s
    | dropPos (s, n) = drop' (expose s, n)
  and drop' (Empty, _) = raise Subscript
    | drop' (Cons(x,s), n) = dropPos (s, n-1)

  fun drop (s,n) = if n < 0 then raise Subscript else dropPos (s,n)
```

```
    fun tabulate f = delay (fn () ⇒ tabulate' f)
    and tabulate' f = Cons (f(0), tabulate (fn i ⇒ f(i+1)))

    (* "Append" one stream to another.  Of course, if the first stream
     is infinite, we'll never actually get to the second stream. *)
    fun append (s1,s2) = delay (fn () ⇒ append' (expose s1, s2))
    and append' (Empty, s2) = expose s2
      | append' (Cons(x,s1), s2) = Cons(x, append (s1, s2))
  end;

structure S = Stream(structure BasicStream = BasicStream);

(* A stream is a lazy list *)

(* Other infinite data structures: Lazy tree *)

signature LAZY_BINARY_TREE =
  sig
    type 'a tree
    datatype 'a tree_top = Leaf | Node of 'a * 'a tree * 'a tree

    (* Lazy tree construction and exposure *)
    val delay : (unit → 'a tree_top) → 'a tree
    val expose : 'a tree → 'a tree_top

    (* Eager lazy tree construction *)
    val leaf : 'a tree
    val make : 'a * 'a tree * 'a tree → 'a tree
  end;

structure LazyBinaryTree :> LAZY_BINARY_TREE =
  struct
    datatype 'a tree = LazyBinaryTree of unit → 'a tree_top
    and      'a tree_top = Leaf | Node of 'a * 'a tree * 'a tree

    (* Lazy tree construction and exposure *)
    fun delay (t: (unit → 'a tree_top)): 'a tree = LazyBinaryTree (t)
    fun expose (LazyBinaryTree (t): 'a tree): 'a tree_top = t ()

    (* Eager lazy tree construction *)
    val leaf : 'a tree = LazyBinaryTree (fn () ⇒ Leaf)
    fun make (x: 'a, l: 'a tree, r:'a tree): 'a tree =
      LazyBinaryTree (fn () ⇒ Node (x, l, r))
  end

signature LAZY_TREE =
  sig
    type 'a tree
    datatype 'a tree_top = Leaf | Node of 'a * 'a tree S.stream

    (* Lazy tree construction and exposure *)
    val delay : (unit → 'a tree_top) → 'a tree
    val expose : 'a tree → 'a tree_top

    (* Eager lazy tree construction *)
    val leaf : 'a tree
    val make : 'a * 'a tree S.stream → 'a tree
  end
```

```
structure LazyTree :> LAZY_TREE =
  struct
    datatype 'a tree = LazyTree of (unit → 'a tree_top)
    and      'a tree_top = Leaf | Node of 'a * 'a tree S.stream

    (* Lazy tree construction and exposure *)
    fun delay (t: (unit → 'a tree_top)): 'a tree = LazyTree (t)
    fun expose (LazyTree (t): 'a tree): 'a tree_top = t ()

    (* Eager lazy tree construction *)
    val leaf : 'a tree = LazyTree (fn () ⇒ Leaf)
    fun make (t: 'a * 'a tree S.stream): 'a tree = LazyTree (fn () ⇒ Node t)
  end

structure T = LazyTree;

signature GAME =
  sig
    type configuration

    (* Initial configuration of the game *)
    val initial : configuration

    (* Function used to give a score to a configuration *)
    (* score c > score c' means that c' is a "better"   *)
    (* configuration.                                   *)
    val score : configuration → real

    (* Given a configuration returns all the possible   *)
    (* configuration reachable organized as a tree      *)
    val game : configuration → configuration T.tree
  end

structure Game :> GAME =
  struct
    (* TODO: depends on the game *)
    type configuration = unit

    (* TODO: depends on the game *)
    val initial : configuration = ()

    (* val score : configuration *)
    (* TODO: depends on the game *)
    fun score (_: configuration): real =
      raise Unimplemented

    (* TODO: depends on the game *)
    (* val moves : configuration -> configuration S.stream *)
    (* Given a configuration returns all possible next     *)
    (* configurations                                      *)
    fun moves (_: configuration): configuration S.stream =
      raise Unimplemented

    (* val game : configuration -> configuration T.tree *)
    (* Given a configuration returns all the possible   *)
    (* configuration reachable organized as a tree      *)
    fun game (c: configuration) : configuration T.tree =
      let
        val m = moves c
```

```
      in
        T.delay (fn () ⇒ T.Node (c, S.map game m))
      end
  end

signature PLAYER =
  sig
    type configuration (* parameter *)

    exception No_moves

    (* Makes a move *)
    val move : configuration → configuration
  end

functor Player(structure G : GAME)
    :> PLAYER where type configuration = G.configuration =
  struct
    type configuration = G.configuration
    type 'a tree = 'a T.tree
    type 'a tree_top = 'a T.tree_top

    (* This represents a scored configuration *)
    type scored = configuration * real

    exception No_moves

    (* val depth: int                                  *)
    (* Depth of the tree of future moves to look at *)
    val depth: int = 8

    (* val threshold: real                            *)
    (* Stop as soon as configuration with a score *)
    (* higher than the threshold is found             *)
    val threshold: real = 0.7

    (* val is_empty : 'a S.stream -> bool  *)
    (* is_empty returns true if s is empty *)
    fun is_empty (s: 'a S.stream): bool =
        is_empty' (S.expose s)
    and is_empty' (S.Empty: 'a S.front): bool = true
    |   is_empty' (_: 'a S.front): bool = false

    (* val average : real S.stream -> real                    *)
    (* Computes the average of a stream of reals. If the   *)
    (* stream is empty it returns 0.0                         *)
    (*                                                        *)
    (* Invariant: the stream is finite                        *)
    (*                                                        *)
    (* It uses the helper function:                           *)
    (*   val average' : real S.front * real * int -> real  *)
    (*   average' s sum count computes the average of the  *)
    (*   values in the stream assuming you have seen count *)
    (*   elements which sum to the value sum before s.     *)
    (*                                                        *)
    (*   Invariant: the stream is not empty                *)
    (*   Invariant: the stream is finite                   *)
    fun average (s: real S.stream): real =
      let
```

```
        fun average' (S.Empty: real S.front, sum: real, count: int): real =
            sum / (real count)
        |   average' (S.Cons (x, s): real S.front, sum: real, count: int): real
=
            average' (S.expose s, sum + x, count + 1)
      in
        if is_empty s then
          0.0
        else
          average' (S.expose s, 0.0, 0)
      end

    (* val score : configuration tree -> int -> real       *)
    (* score c d produces a scores for a move by analyzing *)
    (* the possible future up to depth d                   *)
    fun score (c: configuration tree) (d: int): real =
        score' (T.expose c) d
    and score' (T.Leaf: configuration tree_top) (_: int): real =
        ~1.0
    |   score' (T.Node (c, _): configuration tree_top) (0: int): real =
        G.score c
    |   score' (T.Node (c, next): configuration tree_top) (d: int): real =
        let
          val scores = S.map (fn c ⇒ score c (d-1)) next
        in
          0.5 * G.score c + 0.5 * (average scores)
        end

    (* val find : (configuration * real) S.stream -> configuration *)
    (* find n finds the first configuration that is scored higher  *)
    (* than the threshold or the highest so far                    *)
    (*                                                             *)
    (* Invariant: the stream is not empty                          *)
    (*                                                             *)
    (* It uses the helper function:                                *)
    (*   val find' : (configuration * real) S.front ->             *)
    (*               (configuration * real) option ->              *)
    (*               configuration option                          *)
    (*   find' n m looks for a configuration that is either better *)
    (*   than the current best solution m or that is over the      *)
    (*   threshold                                                 *)
    fun find (s: (configuration * real) S.stream): configuration =
      let
        fun find'
            (S.Empty: (configuration * real) S.front)
            (NONE: (configuration * real) option): configuration option =
          NONE
        |   find'
            (S.Empty: (configuration * real) S.front)
            (SOME (c, _): (configuration * real) option): configuration option =
          SOME c
        |   find'
            (S.Cons ((c, r), s): (configuration * real) S.front)
            (NONE: (configuration * real) option): configuration option =
          find' (S.expose s) (SOME (c, r))
        |   find'
            (S.Cons ((c, r), s): (configuration * real) S.front)
            (SOME (c', r'): (configuration * real) option): configuration option
 =
```

```
          let
            val best: configuration * real =
              if r > r' then
                (c, r)
              else
                (c', r')
          in
            if r > threshold then
              SOME c'
            else
              find' (S.expose s) (SOME best)
          end

        val SOME best = find' (S.expose s) NONE
      in
        best
      end

    (* val choose : configuration T.tree_top -> configuration      *)
    (* Chooses a move among the ones given by the tree of possible *)
    (* moves                                                       *)
    (*                                                             *)
    (* It uses the helper function:                                *)
    (*   val top : configuration tree -> configuration             *)
    (*   top t returns the configuration at the top of the tree    *)
    (*                                                             *)
    (*   Effects: raises No_moves if no move is found              *)
    fun choose (T.Leaf: configuration tree_top): configuration =
        raise No_moves
    |   choose (T.Node (c, next): configuration tree_top): configuration =
        let
          fun top (c: configuration tree): configuration =
              top' (T.expose c)
          and top' (T.Leaf: configuration tree_top): configuration =
              raise No_moves
          |   top' (T.Node (c, _): configuration tree_top): configuration =
              c

          val scored = S.map (fn c ⇒ (top c, score c depth)) next
        in
          find scored
        end

    (* val move : configuration -> configuration *)
    (* Makes a move *)
    fun move (c: configuration): configuration =
      let
        val future = G.game c
      in
        choose (T.expose future)
      end
  end
```