Contents

1 Preamble 2
  1.1 The `SHOW` signature 2

2 Signatures 3
  2.1 Game 3
  2.2 Player 3
  2.3 Controller 3
  2.4 Estimator 4

3 Game 5
  3.1 Types 5
  3.2 Functions 5

4 Player 6

5 Controller 7

6 Estimator 8
1 Preamble

The GAME signature encodes the rules for a particular game.

The PLAYER signature encodes a player for a specific game, which amounts to simply its internal state and a function to choose how to make a move.

The CONTROLLER signature encodes an abstract “referee” or “arena” for a given game. It is typically implemented as a functor which keeps track of given players, alternating control as specified by its game.

The ESTIMATOR signature encodes an estimator for a two-team, zero-sum game.

1.1 The SHOW signature

The GAME signature includes four structures encoding the players, moves, states, and outcomes for a particular game, called Move, State, and Outcome. These structures ascribe to the SHOW signature:

```plaintext
signature SHOW =
sig
  type t
  val toString : t -> string
end
```
2  Signatures

2.1  Game

```ocaml
signature GAME =
sig

  structure State : SHOW (* public knowledge *)
  structure Move : SHOW (* moves *)
  structure Outcome : SHOW (* result of the game *)

  datatype status = Playing of State.t | Done of Outcome.t

  exception InvalidMove of string

  val play : State.t * Move.t -> status
  val player : State.t -> Player.t
  val moves : State.t -> Move.t Seq.t

end
```

2.2  Player

```ocaml
signature PLAYER =
sig

  structure Game : GAME

  val next_move : Game.State.t -> Game.Move.t

end
```

2.3  Controller

```ocaml
signature CONTROLLER =
sig

  structure Game : GAME

  val play : Game.State.t -> Game.Outcome.t

end
```
2.4 Estimator

```
signature ESTIMATOR =
sig

structure Game : GAME

type guess
datatype est = Definitely of Game.Outcome.t | Guess of guess

val compare : est * est -> order
val toString : guess -> string

val estimate : Game.State.t -> guess

end
```
3 Game

The provided `structure Player` contains a datatype, representing Minnie and Maxie, and some relevant utility functions.

3.1 Types

- The `Move.t` type represents a move within the game.
- The `State.t` type represents the state of a game. Note that a given state is public information.
- The `Outcome.t` type represents all potential outcomes of an instance of the game.

Starting from a `State.t`, a move is made, which results in a `status`. This will either indicate that the game is still in play (the `Playing` constructor), providing a new state, or indicate that the game is done (the `Done` constructor), providing an outcome.

3.2 Functions

```
play : State.t * Move.t -> status
REQUIRES: s is valid, according to the rules of the game.
ENSURES:

- Suppose m is a valid move for state s, according to the rules of the game. Then, play (s,m) ⇒ st, where st is of the form Playing s' if the game is still in play or Done oc if the game is completed.
- Otherwise, play (s,m) raises InvalidMove err, for some string err.
```

```
player : State.t -> Player.t
REQUIRES: s is valid, according to the rules of the game.
ENSURES: player s evaluates to a value.
```

```
moves : State.t -> Move.t Seq.t
REQUIRES: s is valid, according to the rules of the game.
ENSURES: moves s ⇒ ms, where ms represents all valid moves for state s.
```
4 Player

\[
\text{next\_move} : \text{Game\_State}\_t \rightarrow \text{Game\_Move}\_t
\]

REQUIRES: s is a valid game state.

ENSURES: \text{next\_move} s \Rightarrow m, where m is the desired move to make.
5 Controller

Given a starting state, a controller executes a game to completion, producing an outcome. This should follow the model of players provided by the game, given players.

\[
\text{play : Game.State.t \rightarrow Game.Outcome.t}
\]

REQUIRES: \( s \) is a valid game state which terminates in an outcome, according to Game.

ENSURES: \( \text{play } s \Rightarrow oc \), where \( oc \) is the outcome of playing from \( s \) according to Game.
6 Estimator

The guess type represents a guess. Typically, it will be a numerical quantity, like int.

The est datatype encodes the notion of an estimate, where either the game is finished with an outcome or a guess was made.

\[
\text{compare} : \text{est} \times \text{est} \rightarrow \text{order}
\]
ENSURES: compare forms a total ordering.

\[
\text{toString} : \text{guess} \rightarrow \text{string}
\]
ENSURES: toString g converts a guess to a string representation.

\[
\text{estimate} : \text{Game}.\text{State}.t \rightarrow \text{guess}
\]
ENSURES: estimate s evaluates to a value.

Additionally, a functor MiniMax is included, which takes in a settings structure ascribing to the following signature:

```plaintext
signature SETTINGS =
 sig
  structure Est : ESTIMATOR
  val search_depth : int
end
```