

15–150: Principles of Functional Programming

Implementing Game Playing

Michael Erdmann*

Spring 2020

1 Background

Recall from the previous lecture that we introduced a signature for games:

```
signature GAME =
sig
  datatype player = Minnie | Maxie
  datatype outcome = Winner of player | Draw
  datatype status = Over of outcome | In_play

  type state (* abstract, representing states of the game *)
  type move  (* abstract, representing moves of the game *)

  val start : state

  (* REQUIRES: m is in moves(s) *)
  (* ENSURES:  make_move(s,m) returns a value. *)
  val make_move : state * move -> state

  (* REQUIRES: status(s) == In_play *)
  (* ENSURES:  moves(s) returns a nonempty sequence of moves legal at s. *)
  val moves : state -> move Seq.seq

  val status : state -> status
  val player : state -> player (* says whose turn it is to make a move *)

  datatype est = Definitely of outcome | Guess of int

  (* REQUIRES:  status(s) == In_play *)
  (* ENSURES:  estimate(s) returns a value. *)
  val estimate : state -> est

  [plus helper functions useful for input/output, omitted ]
end
```

*Adapted from documents by Stephen Brookes and Dan Licata.

We say that a move `m` is *legal* at state `s` if `m` is in the sequence `moves(s)`. The specs say that: (i) performing a legal move at a state does return a new state, i.e., it does not raise an exception or loop forever; (ii) at every in-play state there is at least one legal move; and (iii) the estimate function always returns a value when applied to an in-play state.

We also introduced a structure `Nim:GAME` with a perfect estimator and `DumbNim:GAME` with a nearly useless estimator.

Today, we will talk about playing games: We will implement players that use various versions of the minimax algorithm to figure out the best moves to make. We will introduce referees as mechanisms that allow players to interact and actually play games.

2 Players

A player for a game is just a game equipped with a function that picks a next move for any in-play state:

```
signature PLAYER =
sig
  structure Game : GAME      (* parameter *)

  (* REQUIRES: Game.status(s) == In_play *)
  (* ENSURES:  next_move(s) evaluates to a Game move legal at s. *)
  val next_move : Game.state -> Game.move
end
```

Again note the comments in the signature, which we regard as guidance to be followed by all implementations of this signature. Thus, implementors of players should be careful to only ever call the `next_move` function on a state that is in-play for the game in question.

3 Implementing Players

First, we will implement an unbounded (full) minimax algorithm that works in the same way that we labeled the game tree for Nim in the previous lecture. It works in principle for any game for which game trees are finitely deep, although the runtime may still be prohibitive if game trees are big. Then we will modify this algorithm to obtain a depth-bounded minimax player that searches up to a fixed depth and uses an estimator instead of exploring deeper.

3.1 Unbounded Minimax

Let's implement a generic player for games that always uses minimax to decide on the best move to take: this is an *unbounded* use of the minimax algorithm. Instead of building game trees, the algorithm is implemented using *recursive* functions from states to moves and from states to estimates. In figuring out the best move at one state (best for the player whose turn it is), we make recursive calls to find the best moves for the other player at states reachable by single moves. This recursive flow of control mimics the bottom-up propagation of values we discussed previously. Moreover, we will discover that it is very natural to define two functions, one for state evaluation and one for move selection, each of which calls the other; this is known as a pair of *mutually recursive* functions. SML supports mutual recursion.

Keeping track of moves and values: At first cut, one might write a function to evaluate each game state and return its value, propagating the values of reachable states up the game tree, as we did in class. However, at the very top level, one needs to know not only the value of the state, but also the corresponding optimal move. Much of the code needed for move selection is going to resemble the code that implements the evaluation function. To avoid code duplication, we will use a function that calculates this best-move information at any state (even though we may only need it at the initial state): this function computes for each state both its value and the move that ensures that value. We call such a pair (a move and a value) an *edge*, by analogy with the structure of the game tree.

Functors: Since the idea of minimax makes sense for any game, not just for Nim, we encapsulate this idea by means of a functor (Figure 1). Beware, however, that it would be a bad idea to use this functor to build a player for a game with infinite game trees!

reduce1: We will use a variant form of the **reduce** operation on sequences, because we only ever need to use it in this code to take the maximum or minimum over a non-empty sequence. (In-play states always have at least one move out of them.) In cases like this there is no need to supply a base case value to the reduce operation. We have included in the **SEQUENCE** signature the following function:

$$\text{reduce1} : (('a * 'a) \rightarrow 'a) \rightarrow 'a \text{ seq} \rightarrow 'a$$

with the following specification, assuming that \odot is associative: (recall that we use the notation \odot to describe \odot as if it were infix)

$$\text{reduce1 } g \langle x_1, \dots, x_n \rangle \cong x_1 \odot x_2 \odot \dots \odot x_n.$$

Parallelism: Note the opportunities for parallelism in implementing minimax: at each level, one may explore each next state in parallel, and combine the results together, with span *logarithmic* in the number of possible moves, even though the work is *linear* in the number of moves. (That's why we use sequences here rather than lists.)

Estimator values: Recall that **GAME** comes with a type **est** and a function **estimate** which provides an approximation to the potential value of a state. An estimation produces either **Definitely** an outcome, or a **Guess** of an integer; a positive guess indicates “better for Maxie”, whereas a negative guess indicates “better for Minnie”. The magnitude of a guess indicates how favorable the estimate looks. The sign of an estimate for a given state is absolute, i.e., not relative to the player whose turn it is in that state. We regard the type of estimate values as an ordered type, with a greater-than ordering based on the following:

```
Definitely (Winner Maxie)    >
Guess(some positive number) >
Guess(0) and Definitely Draw >
Guess(some negative number) >
Definitely (Winner Minnie)
```

Comments: (a) The unbounded minimax player will never make any guesses, no matter what the game is, but the depth-bounded minimax player will make guesses via its estimator. (b) The estimator appears inside **GAME** for simplicity in these lectures. More generally, one would want to place it in a separate signature/structure.

This ordered type neatly generalizes the “ -1 is less than $+1$ ” fact that we exploited in class last time, when explaining how taking the maximum and minimum labels at a node were appropriate; previously $+1$ was used for “Maxie wins” and -1 for “Minnie wins”. Note that the estimate value `Guess 0` is tantamount to `Definitely Draw`, since these two values occupy the same position in terms of this ordering, and there is no way to gain extra information from one or the other.

In the functor body, the function `lesseq : G.est * G.est -> G.est` implements this ordering, and is used to obtain the relevant `max` and `min` functions for edges, based on estimates. (Here `G` is the current game implementation; it is a structure argument to the functor and it ascribes to signature `GAME`.)

Of course, there is nothing particularly significant about this value space for the estimator. In a different implementation, one might use reals or other datatypes.

Searching and Evaluating

The function `search` takes a state and chooses the max/min edge out of the state (as appropriate for the player in that state). To compute all the edges, we pair each allowed move with the value of the state resulting from the move, as computed by the helper function `evaluate`. To `evaluate` a state, we return the actual outcome if the state is `Over`, or call `search` again, followed by an application of the little helper function `edgeval` that forgets the move component. We started off writing `evaluate` as a helper for `search`, but now `evaluate` calls `search` too. This is an example of what is called *mutual recursion*: two functions, each of which calls the other. This isn’t so different conceptually from a function calling itself recursively, but it is sometimes more convenient to express an algorithm this way. To indicate mutually recursive definitions, one writes `and` instead of `fun` for the second definition. In general, one can declare two, three, \dots , as many as one needs, mutually recursive functions, using `and` to link the definitions, as in

```
fun f1(x1) = e1
and f2(x2) = e2
and f3(x3) = e3
```

To compute a `next_move`, the code calls `search`, then selects the move from the edge that is returned.

Check that this functor definition (Figure 1) fulfills the specification, provided the game `G` has finitely deep game trees.

(If `G` has infinite sequences of legal moves, the `search`, `evaluate`, `next_move` functions may loop forever.)

Also note where we took care to ensure that the `search` function only gets used on in-play game states: the only way for a user of this structure to invoke this function is by calling `next_move`, and the spec for `GAME` requires that this function is only ever used on an in-play state.

To test this functor code, we removed the signature ascriptions (to make the helper functions inside the functor body visible). We also used a structure `Seq` that implements sequences as lists, just to keep things simple for testing purposes. (Once tested, one should again ascribe the structures to their respective signatures and use a parallel-friendly implementation of sequences.) See Figure 2 for a summary of the results. Check that you see the relationship between these results and the shape of the labeled tree for Nim that we built in class last time. As we promised, the bottom-up propagation of labels gets implemented by recursive function calling.

```

functor FullMiniMax (G : GAME) : PLAYER =
struct
  structure Game = G

  type edge = G.move * G.est
  fun edgemove (m,v) = m
  fun edgeval (m,v) = v

  fun lesseq(x, y) = (x=y) orelse
    case (x, y) of
      (G.Definitely(G.Winner G.Minnie), _) => true
    | (_, G.Definitely(G.Winner G.Maxie)) => true
    | (G.Guess n, G.Definitely G.Draw) => (n <= 0)
    | (G.Definitely G.Draw, G.Guess m) => (0 <= m)
    | (G.Guess n, G.Guess m) => (n <= m)
    | (_, _) => false

  (* max : edge * edge -> edge , min : edge * edge -> edge *)
  fun max (e1, e2) = if lesseq(edgeval e2, edgeval e1) then e1 else e2
  fun min (e1, e2) = if lesseq(edgeval e1, edgeval e2) then e1 else e2

  (* choose : G.player -> edge Seq.seq -> edge *)
  fun choose G.Maxie = Seq.reduce1 max
    | choose G.Minnie = Seq.reduce1 min

  (* search : G.state -> edge *)
  (* REQUIRES: status(s) == In_play *)
  fun search (s : G.state) : edge =
    choose (G.player s)
      (Seq.map
        (fn m => (m, evaluate (G.make_move(s,m))))
        (G.moves s))

  (* evaluate : G.state -> G.est *)
  and evaluate (s : G.state) : G.est =
    case G.status s of
      G.Over(v) => G.Definitely(v)
    | G.In_play => edgeval(search s)

  (* recall: the signature requires that s be In_play. *)
  val next_move = edgemove o search
end

```

Figure 1: FullMiniMax

```

structure Nim =
struct ... end (* as last time, but without ascription *)

functor FullMiniMax (G : GAME) =
struct . . . end (* as above, but without ascription *)

structure FMMNimPlayer = FullMiniMax(Nim);
open Nim; (* only opening briefly for debugging; don't do this in submitted code *)
open FMMNimPlayer;

(* Here we use a list implementation of sequences, so that SML shows us
   what is visible, for this document.
   More generally, when you are writing code, you will need to use the
   provided sequence functions to inspect values.
*)

- evaluate(State(3,Maxie));
val it = Definitely (Winner Maxie) : est

- search(State(3, Maxie));
val it = (Move 2, Definitely (Winner Maxie)) : edge

- Seq.map (fn m => (m, evaluate(make_move(State(3, Maxie), m))))
  (moves(State(3, Maxie)));
val it =
  [(Move 1, Definitely (Winner Minnie)),
   (Move 2, Definitely (Winner Maxie)),
   (Move 3, Definitely (Winner Minnie))] : (move * est) Seq.seq

- Seq.reduce1 max it;
val it = (Move 2, Definitely (Winner Maxie)) : move * est

```

Figure 2: Testing FullMiniMax

We can also use extensional equivalence to reason about game behavior. In particular, it follows from the SML definitions that: (here we use the angle-bracket notation for sequences)

```

evaluate(State(3, Maxie))
  ≅ edgeval(search(State(3, Maxie)))
  ≅ edgeval(reduce1 max <(Move 1, evaluate(State(2, Minnie))),
              (Move 2, evaluate(State(1, Minnie))),
              (Move 3, evaluate(State(0, Minnie)))>)

evaluate(State(0, Minnie)) ≅ Definitely(Winner Minnie)

evaluate(State(1, Minnie))
  ≅ edgeval(search(State(1, Minnie)))
  ≅ edgeval(reduce1 min <(Move 1, evaluate(State(0, Maxie)))>)
  ≅ edgeval(Move 1, Definitely (Winner Maxie))
  ≅ Definitely (Winner Maxie)

evaluate(State(2, Minnie))
  ≅ edgeval(search(State(2, Minnie)))
  ≅ edgeval(reduce1 min <(Move 1, evaluate(State(1, Maxie))),
              (Move 2, evaluate(State(0, Maxie)))>)
  ≅ edgeval(reduce1 min <(Move 1, Definitely(Winner Minnie)),
              (Move 2, Definitely(Winner Maxie))>)
  ≅ edgeval(Move 1, Definitely(Winner Minnie))
  ≅ Definitely(Winner Minnie)

```

So we see that

```

evaluate(State(3, Maxie))
  ≅ edgeval(reduce1 max <(Move 1, evaluate(State(2, Minnie))),
              (Move 2, evaluate(State(1, Minnie))),
              (Move 3, evaluate(State(0, Minnie)))>)
  ≅ edgeval(reduce1 max <(Move 1, Definitely(Winner Minnie)),
              (Move 2, Definitely(Winner Maxie)),
              (Move 3, Definitely(Winner Minnie))>)
  ≅ edgeval(Move 2, Definitely(Winner Maxie))
  ≅ Definitely(Winner Maxie)

```

3.2 Depth-Bounded Minimax

Note that if we ask (using the unbounded minimax Nim player from above)

```
- evaluate(State(150, Maxie));
```

the computation takes a long time! Reason: the game tree with root 150 has depth 150, so this function makes an enormous number of recursive calls.

Next, we implement a player that uses minimax to a chosen depth. If there are game states still in-play at that depth, then the player calls the estimator to estimate the state value for such game states (Figure 3). We present this as a functor applicable to an arbitrary **GAME** structure coupled with a chosen search depth parameter. It will be convenient to introduce an extra signature **SETTINGS**, given by:

```
signature SETTINGS =
sig
  structure Game : GAME      (* parameter *)
  val depth : int
end
```

The implementation of bounded minimax is very similar in spirit to the previously presented unbounded minimax. We've deliberately written the code and explanation to facilitate comparison between the two implementations.

The function **search** takes a *depth parameter* and a state and **chooses** the max/min edge out of the state (as appropriate for the player in that state). To compute all the edges, we pair each move with the value of the state resulting from the move, as computed by the helper function **evaluate**, *which also takes a depth parameter*. To **evaluate** a state, we return the outcome if the state is **Over** or *we use the estimator if we're out of depth*. Otherwise, we want to use **search** on the new states, then forget the move component, as before. Again, **evaluate** and **search** are mutually recursive.

To compute a **next_move**, we **search with the depth provided in Settings**, then select the move that is returned.

This functor (Figure 3) fulfills the specification, even if the game tree for **G** is infinitely deep. For nonnegative values of **d**, **search d** returns a value when applied to an in-play state, and **evaluate d** returns a value when applied to a state.


```

functor MiniMax (Settings : SETTINGS) : PLAYER =
struct
  structure Game = Settings.Game

  (* We also abbreviate Game as G, to keep the notation simple below. *)
  structure G = Game

  type edge = G.move * G.est
  fun edgemove (m,v) = m
  fun edgeval (m,v) = v

  fun lesseq(x, y) = (x=y) orelse
    case (x, y) of
      (G.Definitely(G.Winner G.Minnie), _) => true
    | (_, G.Definitely(G.Winner G.Maxie)) => true
    | (G.Guess n, G.Definitely G.Draw) => (n <= 0)
    | (G.Definitely G.Draw, G.Guess m) => (0 <= m)
    | (G.Guess n, G.Guess m) => (n <= m)
    | (_, _) => false

  (* max : edge * edge -> edge , min : edge * edge -> edge *)
  fun max (e1, e2) = if lesseq(edgeval e2, edgeval e1) then e1 else e2
  fun min (e1, e2) = if lesseq(edgeval e1, edgeval e2) then e1 else e2

  (* choose : G.player -> edge Seq.seq -> edge *)
  fun choose G.Maxie = Seq.reduce1 max
    | choose G.Minnie = Seq.reduce1 min

  (* search : int -> G.state -> edge *)
  (* REQUIRES: d > 0, status(s) == In_play *)
  fun search (d : int) (s : G.state) : edge =
    choose (G.player s)
      (Seq.map
        (fn m => (m, evaluate (d - 1) (G.make_move(s,m))))
        (G.moves s))

  (* evaluate : int -> G.state -> G.est *)
  (* REQUIRES: d >= 0. *)
  and evaluate (d : int) (s : G.state) : G.est =
    case (G.status s, d) of
      (G.Over(v), _) => G.Definitely(v)
    | (G.In_play, 0) => G.estimate s
    | (G.In_play, _) => edgeval(search d s)

  (* recall: the signature requires that s be In_play. *)
  val next_move = edgemove o (search Settings.depth)
end

```

Figure 3: Depth-bounded minimax

3.3 Comparing Strategies

We can observe some differences in behavior that illustrate the way players behave when using different strategies.

Minimax using the dumb estimator, searching to depth 4:

```
(* minimax up to depth 4, then uninformative guesses *)
structure MM4DumbNimPlayer = MiniMax(struct structure Game = DumbNim  val depth = 4 end)
MM4DumbNimPlayer.next_move DumbNim.start
```

This strategy picks Maxie's first move, at the state consisting of 15 pieces, to be `Move 1`. This is definitely not good! A smart player can win when presented with 14 pieces!

Why does Maxie make this choice? (The game tree depth is more than 4, so Maxie computes the maximum of edges (`Move 1, Guess 0`), (`Move 2, Guess 0`) and (`Move 3, Guess 0`). This computation yields `Move(1, Guess 0)`, so the chosen move is `Move 1`, by the way `max` and `reduce1` are implemented.)

Minimax using the dumb estimator, with unbounded search:

```
(* full minimax, never guesses *)
structure FMMDumbNimPlayer = FullMiniMax(DumbNim)
FMMDumbNimPlayer.next_move DumbNim.start
```

This strategy picks Maxie's first move to be `Move 2` – the genius move! To be expected, since the search tree is finite and we have full use of minimax. The estimator is irrelevant.

Minimax using the smart estimator, searching to depth 4:

```
(* minimax up to depth 4, then perfect estimation *)
structure MM4NimPlayer = MiniMax(struct structure Game = Nim  val depth = 4 end)
MM4NimPlayer.next_move Nim.start
```

This strategy picks Maxie's first move to be `Move 2` – again the genius move. Reason: even though the game tree is deeper than 4 levels, the estimator in the `Nim` structure is smart enough to supply useful information (in fact, perfect)!

4 Referees and Tournaments

A tournament is a game, played by two players, starting from the initial state of the game.

4.1 Terse Referee

The simplest kind of referee that would make sense for two-person tournaments is one that takes two players, starts at the initial state, and keeps letting the player whose turn it is choose the next move, until the game is over, whereupon the referee prints a string indicating the outcome. (A more verbose referee might also produce a running commentary by printing play-by-play information throughout the tournament.)

To help with implementation, we augment the `GAME` signature with a function

```
outcome_to_string : outcome -> string
```

that turns a game outcome into a string. And then we need to augment the game structures (e.g., `Nim` and `DumbNim`) so that they also implement this operation. Let's assume we've done this:

```
fun outcome_to_string (Winner Maxie) = "Maxie wins!"  
  | outcome_to_string (Winner Minnie) = "Minnie wins!"  
  | outcome_to_string (Draw) = "Game tied!"
```

As defined in Figure 4, the referee for a given game uses the game's `outcome_to_string` function to generate a “final outcome” string, and uses the built-in SML function

```
print : string -> unit
```

to print that string to the screen.¹

The previous description is almost enough, but there is a problem! Implicitly we assumed that the two players were playing the same game. And the referee needs to be able to use the `next_move` functions of the two players, on the same type of state. But for all we know, Maxie might be playing Connect 4 and Minnie playing chess, thinking they are playing the same game! To fix this, we use a *sharing constraint*: we define a signature for *two players playing the same game*, with sharing constraints that require the two player structures to have the same types of game states and the same types of game moves. The effect is that any structure with this signature will involve two players for which the referee can indeed supervise. (Sharing constraints are a feature of the SML module system that we see here for the first time but won't discuss in detail or use much again. Sharing constraints let you demand coherence between structures, which can be important for combining different modules together into larger pieces. We could also replace the two `sharing type` constraints with the single constraint `sharing Maxie.Game = Minnie.Game` in this example.)

```
signature TWO_PLAYERS =  
sig  
  structure Maxie : PLAYER      (* parameter *)  
  structure Minnie : PLAYER     (* parameter *)  
  sharing type Maxie.Game.state = Minnie.Game.state  
  sharing type Maxie.Game.move = Minnie.Game.move  
end
```

¹The `print` function, and several other useful input/output primitives, are implemented in a structure `TextIO`.

```

functor Referee (P : TWO_PLAYERS) : sig val go : unit -> unit end =
struct
  structure G = P.Maxie.Game
  structure H = P.Minnie.Game

  (* run: G.state -> string *)
  fun run s =
    case (G.status s, G.player s) of
      (G.Over(v), _) => G.outcome_to_string(v)
    | (G.In_play, G.Maxie) => run(G.make_move(s, P.Maxie.next_move s))
    | (G.In_play, G.Minnie) => run(H.make_move(s, P.Minnie.next_move s))

  fun go () = print(run(G.start) ^ "\n")
end

```

Figure 4: A simple Referee.

Given the sharing constraints, the referee's body (Figure 4) type checks, because it may assume that the `state` and `move` types are the same for the two players. (If we tried without the sharing constraints, SML would give us a type error.)

To implement `TWO_PLAYERS`, we have to give definitions for each component, i.e., structures for each player, and we must ensure that the sharing constraints hold. We don't need to type any extra lines of code here; SML will figure out if the sharing constraints are valid, given the structure definitions that we supply.

The referee takes two players and produces a function `go : unit -> unit` that lets the players play their game, beginning from the initial state, then prints out the final outcome. Because `unit` is a type with just one value, one sees from the type of `go` that we must be using this function for some kind of effect (in this case, looping and printing), not for producing a value.

A terse referee appears in Figure 4. Here is an example of its use:

```

- structure S = FullMiniMax(Nim);
- structure R =
  Referee(struct structure Maxie = S structure Minnie = S end);
- R.go ();
Maxie wins!
val it = ( ) : unit

```

4.2 A More Verbose Referee

To implement a more verbose referee we would need to equip games with additional helper functions such as:

```

val player_to_string : player -> string
val move_to_string : move -> string
val state_to_string : state -> string

```

and we would then extend the game structures to implement these too. For example, we might augment Nim with

```

fun player_to_string Maxie = "Maxie"
  | player_to_string Minnie = "Minnie"

fun state_to_string (State (n, p)) =
  Int.toString n ^ " pieces left, and " ^ player_to_string p ^ "'s turn"

fun move_to_string (Move k) = Int.toString k

```

An functor implementation of this chatty referee appears in Figure 5.

Here are some results that illustrate the referee's behavior.

```

- structure S = FullMiniMax(Nim);
- structure R =
  VerboseReferee(struct structure Maxie = S structure Minnie = S end);
- R.go ( );
15 pieces left, and Maxie's turn
The move is 2
13 pieces left, and Minnie's turn
The move is 1
12 pieces left, and Maxie's turn
The move is 3
9 pieces left, and Minnie's turn
The move is 1
8 pieces left, and Maxie's turn
The move is 3
5 pieces left, and Minnie's turn
The move is 1
4 pieces left, and Maxie's turn
The move is 3
1 pieces left, and Minnie's turn
The move is 1
0 pieces left, and Maxie's turn
Maxie wins!
val it = () : unit

```

This is a more chatty blow-by-blow account of the same game run as before.

```

functor VerboseReferee (P : TWO_PLAYERS) : sig  val go : unit -> unit end =
struct
  structure G = P.Maxie.Game

  (* play : state -> unit *)
  fun play s =
    case G.status s of
      G.Over(v) => print(G.outcome_to_string(v) ^ "\n")
    | G.In_play =>
      let
        val (s_to_string, m_to_string, next_move, make_move) =
          (case G.player s of
             G.Maxie =>
              (P.Maxie.Game.state_to_string, P.Maxie.Game.move_to_string,
               P.Maxie.next_move, P.Maxie.Game.make_move)
           | G.Minnie =>
              (P.Minnie.Game.state_to_string, P.Minnie.Game.move_to_string,
               P.Minnie.next_move, P.Minnie.Game.make_move))
        val m = next_move(s)
        val s' = make_move(s, m)
      in
        (print ("The move is " ^ m_to_string m ^ "\n");
         print (s_to_string s' ^ "\n");
         play s')
      end

  fun go () =
    let
      val start = G.start
      val _ = print((case G.player start of
                       G.Maxie => P.Maxie.Game.state_to_string
                     | G.Minnie => P.Minnie.Game.state_to_string) start ^ "\n")
    in
      play start
    end
end
end

```

Figure 5: A more verbose referee.

As a final example, you might also like to see what happens when this referee plays two `DumbNim` players (using minimax but with different search depth) against each other. In fact, you might want to predict the likely results. For example:

```
structure DumbNim4v8Tournament =
  VerboseReferee
  (struct
    structure Maxie = MiniMax(struct structure Game = DumbNim  val depth = 4 end)
    structure Minnie = MiniMax(struct structure Game = DumbNim  val depth = 8 end)
  end)

- DumbNim4v8Tournament.go();
15 pieces left, and Maxie's turn
The move is 1
14 pieces left, and Minnie's turn
The move is 1
13 pieces left, and Maxie's turn
The move is 1
12 pieces left, and Minnie's turn
The move is 3
9 pieces left, and Maxie's turn
The move is 1
8 pieces left, and Minnie's turn
The move is 3
5 pieces left, and Maxie's turn
The move is 1
4 pieces left, and Minnie's turn
The move is 3
1 pieces left, and Maxie's turn
The move is 1
0 pieces left, and Minnie's turn
Minnie wins!
val it = () : unit
```

5 Human Player

To implement a “human” player, one that reads input and interprets it as instructions on how to make moves, we need to further extend the signature of games with an I/O function:

```
parse_move : state -> string -> move option
```

We then need to extend the game structure definitions to include such parsers. For instance, in Nim we could include:

```
fun parse_move (State (n, _)) str =
  let
    fun enough k = if k <= n then SOME(Move k) else NONE
  in
    case str of
      "1" => enough 1
    | "2" => enough 2
    | "3" => enough 3
    | _   => NONE
  end
```

If the user types in a string `str`, `parse_move state str` checks whether the string `str` represents an appropriate number of pieces that can be legally removed given the game state `state`.

```
functor HumanPlayer (G : GAME) : PLAYER =
struct
  structure Game = G

  fun readmove () =
    case TextIO.inputLine TextIO.stdIn of
      NONE => raise Fail "early input termination; aborting"
    | SOME(str) => SOME(String.substring(str, 0, String.size(str)-1))
  (* This strips off a trailing newline character, as required by G.parse_move. *)

  fun parsemove(state, NONE) = NONE
    | parsemove(state, SOME(str)) = G.parse_move state str

  fun player_to_string (G.Maxie) = "Maxie"
    | player_to_string (G.Minnie) = "Minnie"

  fun next_move state =
    let
      val _ = print(player_to_string(G.player state) ^ ", please type your move: ")
    in
      case parsemove(state, readmove()) of
        SOME(m) => m
      | NONE => (print "Something is wrong; bad input or bad move.\n";
        next_move state)
    end
end
end
```

The SML code implementing the “human player” uses some additional components of `TextIO`. First, the code prints a prompt asking for a move. Then it reads a line of input from `stdIn` (the keyboard) and tries to parse that input into a valid move. If this fails, the code prints an error and asks again. If this succeeds, the code returns the move. Since this construction works for an arbitrary game, we use a functor.

Finally, putting it all together, here a human may play Nim against depth-bounded MiniMax:

```
structure NimH = HumanPlayer(Nim)
structure Set3 : SETTINGS =
struct
  structure Game = Nim
  val depth = 3
end
structure Nim3 = MiniMax(Set3)
structure HvMM3 : TWO_PLAYERS =
struct
  structure Maxie = NimH
  structure Minnie = Nim3
end
structure Nim_HvMM3 = VerboseReferee(HvMM3);
Nim_HvMM3.go();
```