In these lectures, we will implement the minimax algorithm to play any 2-player, deterministic, perfect-information, zero-sum game. This algorithm produces an optimal answer (not necessarily in optimal time) assuming the other player plays perfectly. In Homework 10, you will be implementing a more intelligent algorithm, called alpha-beta pruning.

This code is a nice use of modules, and in particular functors for code reuse. Finally, it illustrates some features of ML that we haven’t talked about yet: views, mutual recursion, and sharing declarations.

1 Overview

What is a 2-player, deterministic, perfect-information, zero-sum game? 2-player means, well, it’s played by 2 players who alternate taking turns. Deterministic means each move has a well-defined outcome; there is no randomness. (If you roll dice, the game is not deterministic.) Perfect-information means that, at any given moment, all players know the complete state of the game; there is no hidden information. (Games like poker have hidden information.) Zero-sum means that if I win, you lose, and vice versa—what’s good for me is bad for you, but draws are allowed. Examples include chess, checkers, Connect 4, Mancala, Go and Gomoku.

Let’s look at a game called Nim. In Nim, you start with one pile containing 15 pebbles, and to make a move, you pick up 1, 2, or 3 of them. Whoever picks up the last pebble loses. To phrase it differently, if there are 0 pebbles at the start of your turn, you win. Here’s an example:

I pick up 3 (12 left)
You pick up 3 (9 left)
I pick up 2 (7 left)
You pick up 2 (5 left)
I pick up 1 (4 left)
You pick up 3 (1 left)
I pick up 1 (0 left)
You win!

Nim has a winning strategy for whoever goes first. Let’s consider Nim with 5 pebbles. If it’s your turn, and there are 5 pebbles left, then you lose: go ahead and try it. If you take 1, I take 3, and

*based on notes by Brandon Bohrer, Michael Erdmann and others.

1There are many versions of Nim. This one is sometimes called the subtraction game
there is 1 left. If you take 2, I take 2. If you take 3, I take 1. No matter what you do, there is 1
left on your next turn. Similarly, I can reduce 9 to 5, 13 to 9, etc.

What’s the pattern? If it’s your turn, and pebbles mod 4 \(\equiv 1\), then I have a winning strategy.
So I choose my move to always leave the number of pebbles congruent to 1 mod 4.

Nim is special, in that I can do a quick calculation that tells me who will win. For chess, you
can’t tell (in constant time) just by looking at the board who will win. So, if you were writing
a program to play it, what would you do? Use your computational resources to explore possible
future states!

We build a game tree where the nodes are states, the edges are moves, and each row is labeled
with whose turn it is. For example, for Nim starting with 3 pebbles,

```
3      Maxie
 /     /     \
0 1 2  Minnie
 /     /     \
0 0 1  Maxie
     /     \
0  Minnie
```

Next, we assign each node a value, which tells you who wins. We’ll call the two players Maxie
and Minnie; the value is 1 if Maxie wins, and \(-1\) if Minnie wins. We start by labeling the leaves:
if there are 0 pebbles left, and it’s my turn, then you took the last one, so I won.

```
3      Maxie
 /      /     \
(0,-1) (1,?) (2,?) Minnie
     /     /     \
(0,1)  (0,1) (1,?) Maxie
          /     \
(0,-1) Minnie
```

Next, we propagate these labels up the tree. If it’s Maxie’s turn, the value is the maximum
value of the children, because Maxie will choose the maximizing move. If it’s Minnie’s turn, the
value is the minimum instead.

First level:

```
3      Maxie
 /      /     \
(0,-1) (1,?) (2,?) Minnie
     /     /     \
(0,1)  (0,1) (1,-1) Maxie
          /     \
(0,-1) Minnie
```
The propagation on the rightmost subtree says that if there are 2 pebbles left, then Minnie should take 1, leaving 1, rather than 2, leaving 0.

Next level:

\[
\begin{array}{c}
(3,1) \\
/ \ \\
(0,-1) (1,1) (2,-1) Minnie \\
| / \\
(0,1) (0,1) (1,-1) Maxie \\
| \\
(0,-1) Minnie
\end{array}
\]

This says that Maxie should take 2, leaving 1, rather than taking 3 or 1.

This is the minimax algorithm; it computes the value of a game state assuming both players will play optimally. It does not account for things like “that chess board is more confusing, so I think you’ll make a mistake.”

Of course, in a game with a bigger search space, we cannot draw out the whole tree! For example in Nim starting with \( n \) pebbles, we have \( O(3^n) \) states. For an even more extreme example, if we have an \( n \times n \) Go board, we have about \( O((n^2)!)) \) states, also known as: a lot.

Instead, we’ll write a heuristic that looks at the board and approximates its value. For Nim, a perfect heuristic is: is the number of pebbles congruent to 1 mod 4? For chess, a heuristic would include which pieces are left, where they are positioned, etc. This is where the smarts in playing a particular game come in. Then, the overall algorithm for selecting a move is to (a) explore the game tree up to a certain depth and (b) use the heuristic to approximate the value when that depth is reached.

## 2 Game Architecture

The process of game tree search is independent of the particular game. Moreover, the process of putting together a run of a game, given two players, is independent of the game and the players. We represent this by defining some signatures:

```signature GAME
signature PLAYER
```

We can define various \textsc{Game}s, like Chess, Connect 4, Mancala, Othello. We can define various players, like minimax, alpha-beta pruning, and even human players (which prompt the user for a move). Each player works for any game. And we can define a generic referee that puts two
players together and runs a game. This is an example of modular program design, where we will use functors to achieve good code reuse. Moreover, it’s a nice application-specific use of modules.

3 Games

Let’s start with the signature for a game:

signature GAME =
  sig
    datatype player = Minnie | Maxie

type state (* state of the game; e.g. board and player *)
type move (* moves *)

datatype outcome = Winner of player
datatype status = Over of outcome | InPlay

(* Get the status and player from a game state *)
val status : state -> status
val player : state -> player

(* Create a string containing the outcome/player information *)
val outcome_to_string : outcome -> string
val player_to_string : player -> string
val state_to_string : state -> string

(* initial state *)
val start : state

(*
  * REQUIRES: m is in moves s
  *ENSURES:  make_move s m =>
  *          a game state corresponding
  *          to s after move m is made
  *)
val make_move : state -> move -> state

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

(*)
A game must specify:

- a datatype of players.
- a datatype `status` that tells you whether or not the game is over, and, if it is, what the outcome was. `outcome` is itself a datatype.
- abstract types `state` (a game state, including the board, whose turn it is, etc.) and `move` (representing an action a player can take).
- There are various pieces of information you can extract from a state: What are the possible next moves (`moves`)? Is the game over (`status`)? How many turns have elapsed (`turn`)? Whose turn is it (`player`)? These are called `views` of the abstract type.
• A game also comes with a start state (√start√) and a transition function that applies a move to a state.

• A game must implement an estimator in the function estimate. The result of the estimator is an estimate, which is either Maxie wins, Minnie wins, or a guess Value. Positive is better for Maxie; negative is better for Minnie. Estimates are ORDERED (implemented by Estimate.compare) as follows:

Max >
Value(some positive number) >
Value(0) >
Value(some negative number) >
Min

• Finally, a game comes with some parsing and printing functions.

For simplicity, make_move (s, m) may assume it is given a valid move in s. This invariant will be satisfied at call sites because next_move generates valid moves, and so does parse_move. Moreover, we assume that we don’t transition a game that is already over.

Here’s an implementation of Nim, eliding the parsing and printing code:

structure Nim : GAME =
struct
    datatype player = Maxie | Minnie
    datatype outcome = Winner of player
    datatype status = Over of outcome | InPlay;
    datatype state = State of int * player
    datatype move = Move of int

    fun status (State s) =
        case s of
        (0, p) => Over (Winner p)
        _ => InPlay

    fun player (State (_, p)) = p

    val start = State (15, Maxie)

    fun make_move (State (pile, player)) (Move n) =
        State (pile - n, flip player)

    fun moves (State (pile, _)) =
        Seq.tabulate (fn x => Move (x + 1)) (Int.min (pile, 3))
fun is_legal_move (State (pile, _)) (Move n) = pile >= n

fun flip p = case p of Maxie => Minnie | Minnie => Maxie

datatype est = MinnieWins | Guess of int | MaxieWins
[utilities on estimates]

fun toEst p = case p of Maxie => MaxieWins | Minnie => MinnieWins
(* see the Sprague-Grundy theorem on Wikipedia *)
fun estimate (State (pile, p)) =
  case (pile mod 4) of
    1 => toEst (flip p)
  | _ => toEst p

[parsing and printing]
end

• To satisfy a datatype declaration in a signature, you put the same datatype declaration in
  the structure. Pretty boring.

• A state is represented by a pair of an integer (how many pebbles are left) and who will make
  the next move. A move is an integer (which must be 1, 2, or 3). These are defined to be
datatypes that aren’t exported, so they are abstract. This way, no one can accidentally make
an invalid state (“there are 17 pebbles left”).

• For moves: if there are fewer than three pebbles, then you can only take at most that many,
  otherwise; you can take 1, 2, or 3. We could have written

    fun moves (State (pile , _, _)) =
      case pile of
        0 => raise Fail "Invariant violation: called when game is over"
      | 1 => Seq.cons (Move 1, Seq.empty())
      | 2 => Seq.cons (Move 1, Seq.cons (Move 2, Seq.empty()))
      | _ => Seq.cons (Move 1,
                   Seq.cons (Move 2,
                              (Seq.cons (Move 3, Seq.empty()))))

    but the above is slicker.

• For status: if there are no pebbles left, then the game is over, and whoever’s turn it is next
  wins, because whoever took the last one loses.

• For player: this is easy because the player is sitting right there in the state. If we forgot
  to put the player in the state, then we wouldn’t be able to implement this function: the
  operations place demands on the implementation of abstract types.
• The start state is 15 pebbles and Maxie’s turn. To apply a move, we just subtract (the move is assumed to be valid, so this works).

• The estimator just calculates mod 4, and says who is definitely going to win.

4 Dumb Nim

Just for contrast, and for later when we implement bounded minimax players, here is a “bad” implementation of the Nim game (with the same implementation of the types and game operations) in which the estimator is almost useless. The code is exactly the same as the structure called Nim above, except for the estimate function.
structure DumbNim : GAME =
struct
  datatype player = Maxie | Minnie
  datatype outcome = Winner of player
  datatype status = Over of outcome | InPlay;

  (* pebbles left; player *)
  datatype state = State of int * player
  (* how many pebbles to pick up *)
  datatype move = Move of int

  fun status (State s) =
    case s of
      (0, p) => Over (Winner p)
    | _ => InPlay

  fun player (State (_, p)) = p

  val start = State (15, Maxie)

  fun make_move (State (pile, player)) (Move n) =
    State (pile - n, flip player)

  fun moves (State (pile, _)) =
    Seq.tabulate (fn x => Move (x + 1)) (Int.min (pile, 3))

  fun is_legal_move (State (pile, _)) (Move n) = pile >= n

  fun flip p = case p of Maxie => Minnie | Minnie => Maxie

  datatype est = MinnieWins | Guess of int | MaxieWins
  [utilities on estimates]

  fun toEst p = case p of Maxie => MaxieWins | Minnie => MinnieWins

  fun estimate (State (1, p)) = toEst (flip p)
    | estimate _ = Guess 0
    (* If there is exactly 1 pebble left, then the player
     whose turn it is must lose. Otherwise, guess a draw. *)
end
5 Views

The functions \texttt{player} and \texttt{status} are examples of \textit{views}: functions that map an abstract type \texttt{(in this case state)} into values of a datatype revealed by the signature, so that one may see it and use it. This kind of pattern-matching on values derived from abstract types is very useful, so let’s look at some other instances of it.

As we have discussed many times, list operations have bad parallel complexity, but the corresponding sequence operations are much better. However, sometimes one may want to write a sequential algorithm \texttt{(e.g., because the inputs aren’t very big, or because no good parallel algorithms are known for the problem)}. Given the sequence interface so far, it is difficult to decompose a sequence as “either empty, or a cons with a head and a tail.” To implement this using the sequence operations we have provided, you have to write code that would surely lose style points, such as:

\begin{verbatim}
  case Seq.length s of
    0 =>
    | _ => (... Seq.hd s) (... Seq.tl s) ...
\end{verbatim}

It would be better to use patterns to bind variables to pieces of \texttt{s}, as in

\begin{verbatim}
  case s of
    Nil => ...
    Cons(x,xs) => ...
\end{verbatim}

but we cannot pattern-match on \texttt{s} if its value belongs to an abstract type.

We can solve this problem using a \textit{view}. This means that we put an appropriate datatype in the signature, along with functions converting sequences to and from this datatype. This allows us to pattern-match on an abstract type, while keeping the actual representation abstract. \texttt{(By analogy, we put datatypes player, outcome, status) in the signature GAME, and this enabled us in writing HumanPlayer to use pattern-matching on Maxie versus Minnie.)}

For sequences we can extend the \texttt{SEQ} signature with the following components to enable viewing a sequence as a list:

\begin{verbatim}
  datatype 'a lview = Nil | Cons of 'a * 'a seq
  val showl : 'a seq -> 'a lview
  val hidel : 'a lview -> 'a seq
\end{verbatim}

\texttt{(* ENSURES showl(hidel v) = v, hidel(showl s) = s *)}
\texttt{(* ENSURES showl s = Nil if s is empty sequence *)}
\texttt{(* ENSURES showl s = Cons(a, s') if nth s 0 = a and s' is tail of s *)}

Because the datatype definition is in the signature, the constructors can be used outside the abstraction boundary. The \texttt{showl} and \texttt{hidel} functions convert between sequences and list views. Here is an example of using this view to perform list-like pattern matching:

\begin{verbatim}
  case Seq.showl s of
    Seq.Nil => ...
    | Seq.Cons (x, s') => ... uses x and s' ...
\end{verbatim}
lview exposes that a sequence is either empty or has a first element and a rest (or tail). The tail is another sequence, not an lview—for efficiency, we don’t want to convert the whole sequence to a list just to peek at the first element. Thus, showl lets you do one level of pattern matching at a time: you can write patterns like Seq.Cons(x, xs) but not Seq.Cons(x, Seq.Nil) (if you want to match sequences with exactly one element).
6 Minimax

What is a player for a game? Just a function that picks a move for any state:

signature PLAYER =
  sig
    structure Game : GAME
      (* assumes game is In_play *)
      val next_move : Game.state -> Game.move
  end

As an example of a PLAYER, we implement minimax. At a first cut, you might write a function
to label each game state with its value, propagating them up the tree. However, at the very top
level, you need to know not only the value of the root, but also which move takes you to the child
giving you that value. To avoid code duplication, we can compute this information at each level
(even though we only need it at the root): we label each node with both its value and the move
that takes you to the child that gives you that value. We call such a thing an edge of the game
tree.

functor MiniMax (Settings : sig structure G : GAME
  val search_depth : int
  end) : PLAYER =
  struct
    structure Game = Settings.G

    type edge = (Game.move * Game.est)
    fun valueOf ((_,value) : edge) = value
    fun moveOf ((move,_) : edge) = move

    fun max ((m1,v1) : edge, (m2,v2) : edge) : edge =
      case Game.compare (v1, v2) of
          LESS => (m2, v2)
        | _ => (m1, v1)

    fun min ((m1,v1) : edge, (m2,v2) : edge) : edge =
      case Game.compare (v1, v2) of
          GREATER => (m2, v2)
        | _ => (m1, v1)

    fun choose (p : Game.player) =
      case p of
          Game.Maxie => SeqUtils.reduce1 max
        | Game.Minnie => SeqUtils.reduce1 min

    fun search (d : int) (s : Game.state) : edge =
      choose (Game.player s)
(Seq.map
  (fn mv => (mv, evaluate (d - 1) (Game.make_move s mv)))
  (Game.moves s))

and evaluate (d : int) (s : Game.state) : Game.est =
  case (Game.status s, d) of
    (Game.Over (Game.Winner (Game.Minnie)), _) => Game.MinnieWins
    | (Game.Over (Game.Winner (Game.Maxie)), _) => Game.MaxieWins
    | (_, 0) => Game.estimate s
    | _ => valueOf (search d s)

fun next_move (s : Game.state) : Game.move =
  moveOf (search Settings.search_depth s)
end

Before getting to the main part of MiniMax, we need some operations on edges. moveOf extracts the corresponding data out of an edge. max and min choose the maximum or minimum edge of two input edges. These operations use the ordering on values provided by Game.Est.compare.

These ingredients allow us to define the function choose that chooses the max or min edge from a sequence, depending on whether the player is Maxie or Minnie.

The function evaluate takes a depth parameter d (how many levels to do minimax before estimating) and a state, and computes the value associated with that state in the game tree. As we discussed last time, this value is either (1) the outcome, if the game is over; or (2) the estimate of the current state, if the depth is 0; or (3) the max/min (as appropriate for the state’s player) of the values of all the child nodes in the game tree.

For case (3), we use another function search which obtains the sequence of possible moves, and for each one, evaluates to depth d-1 the state resulting from making that move at the current state. Then we reduce min or max over the resulting sequence, depending on which player is next. (Seq.reduce1 implements reduce for a non-empty sequence—when a sequence is non-empty, it only needs the combiner operation, not a base case. This is appropriate here because non-terminal states have at least one move out of them.) Finally, we can forget the actual move since all we wanted to do is evaluate the state.

This is an example of what is called mutual recursion: two functions, each of which calls the other. It’s not that different that a function calling itself recursively, but it’s sometimes more convenient to express things this way. To indicate mutually recursive definitions, you write and instead of fun on the second (and third, etc.—you can have as many as you want) ones.

To compute a next_move, do a search with depth Settings.search_depth. Then we forget the value and just return the move.

Note the opportunities for parallelism here: at each level, you can can explore each next state in parallel, and then combine them together with logarithmic (in the number of possible moves) span, even though it is linear work.

Now, we want a human player which asks you for input, and a referee which allows two players to play against each other.
7 Referee

To implement the human player and the referee, we need to be able to do input and output: read from the keyboard and print to the screen. The referee will be responsible for displaying the game states, while the human player is responsible for asking the user what move to make next.

functor Referee (P : TWO_PLAYERS) : sig val go : unit -> unit end =
  struct
    structure Game = P.Maxie.Game
    
    fun outcomeToString oc =
      case oc of
        Game.Winner (Game.Minnie) => "Minnie wins"
      | Game.Winner (Game.Maxie) => "Maxie wins"
    
    fun play state =
      case Game.status state of
        Game.Over outcome =>
          (print ((Game.state_to_string state) ^ "\n");
            print (outcomeToString outcome ^ "\n")
      | Game.In_play =>
        let
          val () = print ((Game.state_to_string state) ^ "\n")
          val move =
            case Game.player state of
              Game.Maxie => P.Maxie.next_move state
            | Game.Minnie => P.Minnie.next_move state
            in
              play (Game.make_move (state,move))
            end
        in
          fun go () = play Game.start
        end
  end

We define TWO_PLAYERS to be a module pairing together two players:

signature TWO_PLAYERS =
  sig
    structure Maxie : PLAYER
    structure Minnie : PLAYER
  end

The referee produces a function go : unit -> unit. Because unit is the type with exactly one value, how can this function do anything interesting? Because we’re running this function for its effects (reading and printing), not for its value! Another example of this is print : string -> unit.
When we’re dealing with I/O, it can be useful to run a function for its effects, and then run some other function afterwards. Because ML is call-by-value, we already have the tools to sequence execution; for example,

```
let _ = print "Hello! I am about to run a slow function.\n"
in
  fib 100
end
```

("\n" is a newline character.) There’s also a built-in operator ; which runs one expression, throws away its result, and returns the next expression’s result. So we can rewrite this as:

```
print "Hello! I am about to run a slow function.\n"; fib 100
```

With that out of the way, how does the referee work? It checks the status of the game. If it’s over, it prints the state and the final score. If it’s in play, it prints the state, asks the appropriate player to choose a next move, prints the move, and then keeps playing on the resulting state. Overall, we just play from the start state.

**Sharing constraints** The above code seems reasonable enough, but you will get a type error! The reason: it calls Minnie’s next_move function on the result of Maxie’s move, and Maxie’s next_move function on the result of Minnie’s move. But for all we know, Maxie might be playing Nim, and Minnie playing Chess! To fix this, we need a sharing constraint in the functor argument:

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

These constraints say that the states and moves of each player’s games are the same. Given these constraints, the referee’s body type checks, because it may assume that these two types are the same. To implement TWO_PLAYERS, we have to give definitions for each component, starting with structures for each player. What definition do you give for a sharing constraint? Nothing! The constraint is satisfied as long as it is true, given the other definitions. So

```
structure Nim_HvMM =
Referee(struct
  structure Maxie = HumanPlayer(Nim)
  structure Minnie = MiniMax(struct
    structure G = Nim
    val search_depth = 5
  end)
end)
```

satisfies that signature, because MiniMax’s game is its argument’s game, and this is the same in both cases.

Sharing constraints let you demand coherence between structures after the fact, which is important for combining different modules together into larger pieces.
8 Human Player

For the sake of completeness, here is the human player:

functor HumanPlayer (G : GAME) : PLAYER =
struct
  structure Game = G
  
  fun next_move state =
    let val () =
      (print ((case (Game.player state) of
                Game.Maxie => "Maxie"
              | Game.Minnie => "Minnie")
            ^ ", please type your move: ")
    in
      case TextIO.inputLine TextIO.stdIn of
        NONE => raise Fail "Failed to read a line of input from stdin"
      | SOME input =>
        let val input =
          (* eat the newline character from the end of input *)
          String.substring(input, 0, String.size(input) - 1)
        in
          case Game.parse_move state input of
            SOME m => m
          | NONE =>
            let val () = print ("Bad move for this state: "
                               ^ input ^ "\n")
            in next_move state end
    end
  end
end

First, it prints a prompt asking for a move. Then, it reads a line of input from TextIO.stdIn. Assuming this succeeds, it eats the newline character from the end, and then uses the game's parse_move to parse the move. parse_move is given the current state, and is also responsible for checking that the move is valid in the state. If this fails, it prints an error and recurs, to ask again. If it succeeds, it returns the move.

And that’s it! A complete implementation of game playing, including Nim and MiniMax. For homework, you will implement an estimator for Checkers, and the Alpha-Beta algorithm.