In these lectures, we will implement a program that plays any 2-player, deterministic, perfect-information, zero-sum game. The minimax algorithm for playing such games is a standard example in AI. Additionally, 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; sharing declarations; and input/output.

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 (roll the dice). Perfect-information means that, at any given moment, all players know the complete state of the game; there is no hidden information (poker is not perfect-info). 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 (which you’ll do for homework), Mancala, Gomoku.

Let’s illustrate with Nim: you start with 15 pebbles, and to make a move, you pick up 1, 2, or 3 of them. Whoever picks up the last pebble loses.

Here’s an example:

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

In fact, Nim has a winning strategy for whoever goes first: 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 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 ≡ 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 and the edges are moves, and each row is labeled with whose turn it is:

E.g. for Nim, starting from 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, and Maxie wins if the value is 1, and Minnie if it’s −1.

First, we label 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 it’s the min.

First level:

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

Next level:

```
3 Maxie
/ | \ 
(0,-1) (1,1) (2,-1) Minnie
| / \ 
(0,1) (0,1) (1,-1) Maxie
| 
(0,-1) Minnie
```
For example, 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:

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

Maxie

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

Exercise: draw and label the tree starting from 5 pebbles.

What this *minimax algorithm* computes is 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! Instead, we’ll write a heuristic that looks at the board and approximates its value. For Nim, a good (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 doing 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 GAMEs, like Chess, Connect 4, Mancala, Othello. We can define various players, like the above minimax, or other search algorithms that you’ll do for homework, which prune some of the search space when they know it won’t be chosen, or a human player that reads the move as input. 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 =

\[
\text{sig} \\
\text{datatype player} = \text{Minnie | Maxie}
\]
datatype outcome = Winner of player | Draw
datatype status = Over of outcome | In_play

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

(* views of the state: *)
val moves : state -> move Seq.seq (* assumes state is not Over; generates moves that are valid in that state; always generates at least one move *)

val status : state -> status
val player : state -> player

(* initial state and transitions: *)
val start : state
val make_move : (state * move) -> state (* assumes move is valid in that state *)

(* The sign of a guess is absolute, rather than relative to whose turn it is: negative values are better for Minnie and positive values are better for Maxie. *)
datatype est = Definitely of outcome | Guess of int (* estimate the value of the state, which is assumed to be In_play *)
val estimate : state -> est

val move_to_string : move -> string
val state_to_string : state -> string
val parse_move : state -> string -> move option (* ensures move is valid in that state; string is a single line, and *not* newline terminated *)

end

A game must specify:

• a datatype of players. Note that this is a new feature of signatures: you can put a datatype definition in a signature, in which case clients have access to the constructors (which can be used both to create values and for pattern-matching, as usual).

• a datatype status that tells you whether or not the game is over, and, if it is, what the value was. The value is an outcome, which is again specified by 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)? Whose turn is it (player)? These are called views of the abstract type.
• A game also comes with a start state (state) and a transition function that applies a move to a state (cf. finite state automata).

• A game comes with an estimator, which approximates the value of a state; positive is better for Maxie, whereas negative is better for Minnie. The result of the estimator is an estimate, which is either Definitely an outcome, or a Guess. Estimates are ORDERED (implemented by EstOrder) as follows:

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

• 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.

Multiple abstract types at once Note that this signature defines two abstract types, for states and moves, at once. The implementation must be privy to both at once, and clients don’t need to know either. This is perfectly natural in ML, but just try to do it in Java. This is another example like points and vectors in Barnes-Hut.

3.1 Nim

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 | Draw

datatype status = Over of outcome | In_play;

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

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

fun status (State s) =
  case s of
    (0, p) => Over (Winner p)
  | _ => In_play
fun moves (State (pile, _)) = 
    Seq.tabulate (fn x => Move (x + 1)) (Int.min (pile, 3))

val start = State (15, Maxie)

fun flip p = case p of Maxie => Minnie | Minnie => Maxie
fun make_move (State (pile, player), Move n) = 
    case (pile >= n) of 
      true => State (pile - n, flip player) 
    | false => raise Fail "tried to make an illegal move"

(* cf. SpragueGrundy theorem *)
datatype est = Definitely of outcome | Guess of int
fun estimate (State (pile, p)) = 
    case (pile mod 4) of 
      1 => Definitely (Winner (flip p)) 
    | _ => Definitely (Winner 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 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 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 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 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

Next, we implement minimax (Figure 1). At a first cut, you might write a function to label each game state with its value, propagating them up the tree, as above. 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.

Before getting to the main code, we need some operations on edges. First, we define max and min: to implement these operations, we use some type class instances, which say that edges are ordered by their values (ignoring the move), and that max and min are defined for any ordered type. SeqUtils.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. 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 search takes a depth parameter (how many levels to do minimax before estimating) and a state and chooses the max/min (as appropriate for the state’s player) edge out of the 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. To evaluate, we return the actual outcome if the state is Over, or the estimation if we’re out of depth. Otherwise, we want to take the min/max of the values of all children. To avoid code duplication, we can use search to do this! And then apply 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. 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, search with the search_depth that is provided as an argument to the functor, and then select the move that is returned.
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 edgeval (((_,value) : edge) = value
  fun edgemove (((move,_)) : edge) = move

  structure EdgeUtils : sig
    (* ordered by the estimate, ignoring the move *)
    val min : edge * edge -> edge
    val max : edge * edge -> edge
  end
  OrderUtils(PairSecondOrder(struct type left = Game.move
    structure Right = EstOrder(Game)
  end))

  (* assume seqs are non-empty *)
  val choose : Game.player -> edge Seq.seq -> edge =
    fn Game.Maxie => SeqUtils.reduce1 EdgeUtils.max
    | Game.Minnie => SeqUtils.reduce1 EdgeUtils.min

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

  and evaluate (depth : int) (s : Game.state) : Game.est =
    case Game.status s of
      Game.Over v => Game.Definitely v
      | Game.In_play =>
        (case depth of
          0 => Game.estimate s
        | _ => edgeval(search depth s))

  val next_move = edgemove o (search Settings.search_depth)
end

Figure 1: Minimax
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.

Below, we will implement a human player, that plays by asking you for input.

## 5 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. Here are functions we’ll need:

```ocaml
structure TextIO : sig
  type instream
  val stdIn : instream
  val inputLine : instream -> string option
  val print : string -> unit
  ...
end
```

`instream` represents something you can read from; for now, all we will need is `stdin`. `inputLine` reads a newline-terminated string from such a stream. `print` displays characters to standard out. You can also do file I/O using `TextIO`.

Here’s the referee (Fig 2).

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

```ocaml
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, it is clear that we are running this program for its effects (reading and printing), not for its value.

The referee checks the status of the game. If it’s over, it prints the state and the final score. Note that "\n" is a newline character. 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 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 Maxie’s game state, and Maxie’s make move function on Minnie’s move. But for all we know, Maxie might be playing Connect 4, and Minnie chess! To fix this, we need a sharing constraint in the functor argument:

```ocaml
signature TWO_PLAYERS =
sig
  structure Maxie : PLAYER
  structure Minnie : PLAYER
```

9
functor Referee (P : TWO_PLAYERS) : sig val go : unit -> unit end =
  struct
    structure Game = P.Maxie.Game
    structure ShowE = ShowEst(Game)
    val outcomeToString = ShowE.toString o Game.Definitely

    fun play state =
      case Game.status state of
        Game.Over outcome =>
          let val () = print ((Game.state_to_string state) ^ "\n")
              val () = print ("Game over! " ^ outcomeToString outcome ^ "\n")
          in
            ()
          end
        | 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
            val () = print ((case Game.player state of
                  Game.Maxie => "Maxie"
                  | Game.Minnie => "Minnie")
                  ^ " decides to make the move "
                  ^ (Game.move_to_string move)
            )
          in
            play (Game.make_move (state,move))
          end
      end

    fun go () = play (Game.start)
  end

Figure 2: Referee
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.

6 Human Player

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

First, it prints a prompt asking for a move. Then, it reads a line of input from standard in. 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 Connect 4 and the Jamboree algorithm for parallel game tree search with pruning.

7 Views

The functions player and status are examples of views: functions that map an abstract type (in this case state) into a datatype, so that you can pattern-match on it. This of pattern-matching on abstract types is very useful, so let's look at some other instances of it.

As we have discussed many times, lists operations have bad parallel complexity, but the corresponding sequence operations are much better. However, sometimes you want to write a sequential algorithm (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 lose style points:

```ml
case Seq.length s of
  0 =>
  | _ => ... (Seq.hd s) and (Seq.tl s) ...
```

It would be better to have the test function directly return whatever it proves to exist (avoid boolean blindness!):

```ml
case s of
  Nil => ...
  | Cons(x,xs) => ...
```

But we cannot directly pattern-match on s because it is an abstract type.

We can solve this problem using a view. This means 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. We extend the SEQUENCE signature with the following components to enable viewing a sequence as a list:
datatype 'a lview = Nil | Cons of 'a * 'a seq

val showl : 'a seq -> 'a lview
val hidel : 'a lview -> 'a seq

(* invariant: showl (hidel v) ==> v *)

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

case Seq.showl s of
  Seq.Nil => ... (* Nil case *)
| Seq.Cons (x, s') => ... uses x and s' ... (* Cons case *)

lview exposes that a sequence is either empty or has a first element and a rest. The rest 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) (to match a sequence with exactly one element).

We have also provided hidel, which converts a view back to a sequence—Seq.hidel (Seq.Cons(x,xs)) is equivalent to Seq.cons(x,xs) and Seq.hidel Seq.nil is equivalent to Seq.empty(). This can be used to state an induction principle for sequences, enabling us to reason about sequences using induction, as if they were lists or tree. For example:

Consider a predicate \( P(s:'a Seq.seq) \) on sequences.
To prove \( \forall s:'a Seq.seq. P(s) \), it suffices to show

- \( P(Seq.hidel Seq.Nil) \)
- For all \( x:'a \) and \( xs:'a Seq.seq \) if \( P(xs) \) then \( P(Seq.hidel (Seq.Cons x xs)) \)

SEQUENCE also provides a tree view:

datatype 'a tview = Empty | Leaf of 'a | Node of 'a seq * 'a seq

val showt : 'a seq -> 'a tview
val hidet : 'a tview -> 'a seq

that lets you pattern-match on a sequence as a tree. Such a pattern-match is essentially a Seq.mapreduce, but sometimes it is nice to write in pattern-matching style.