We implement a program that plays a 2-player, deterministic, finitely branching, zero-sum game of perfect information. (Yes, we’ll explain what all those terms mean, and give examples!)

The minimax algorithm for playing such games is a standard example in AI, and we’ll implement it in ML. Our code development provides a nice use of modules, and in particular an elegant way to use functors to re-use code. Furthermore, we illustrate some features of ML that we haven’t talked about yet, notably mutual recursion and value-passing exceptions.

*Building on notes by Dan Licata*
1 Overview

Games

What is a 2-player, deterministic, perfect-information, finitely branching, zero-sum game?

- “2-player” means the game played by 2 players who alternate taking turns (well, obviously!).
- Determinism means that each move has a well-defined outcome; there is no randomness (no luck, no rolling the dice).
- Perfect-information means that, at any given moment, both players know the complete state of the game; there is no hidden information.
- Finitely branching means that there are only a finite number of allowed moves at each stage of the game.
- Zero-sum means that if I win, you lose, and vice versa—what’s good for me is bad for you; but draws may be allowed, even in zero-sum games.

Examples include tic-tac-toe\(^1\), Nim, chess, checkers, Connect 4, Mancala, and Gomoku. By way of contrast, poker is not a perfect-information game, because neither player knows the cards held by the other.

Nim

Let’s illustrate with Nim: the game (usually) starts with 15 matchsticks; to make a move, whoever’s turn it is picks up 1, 2, or 3 of them. The player who picks up the last matchstick(s) loses \(^2\).

Here’s an example of a run of Nim, a sequence of plays starting from the initial game state. Remember, we start with 15 matchsticks. The comments below describe a run of the game (since you may not know the rules, I’ll start), interspersed with a running commentary giving information about the game state after each move:

\(^1\)Known as noughts and crosses in Britain, for all you Anglophones.

\(^2\)There are many variants of Nim, including misère forms in which the player who picks up the final matchstick(s) is deemed the winner. In fact, Nim and this misère version form an example of a pair of dual games.
Start of game (15 left)
I pick up 2 (13 left)
You pick up 3 (10 left)
I pick up 1 (9 left)
You pick up 1 (8 left)
I pick up 3 (5 left)
You pick up 3 (2 left)
I pick up 1 (1 left)
You pick up 1 (0 left, and you lose!)

Actually I knew all along how to win! For the Nim game starting from 15 matchsticks there is a winning strategy for whoever goes first. Let’s explain how.

For simplicity, let’s consider Nim starting with 5 matchsticks. If it’s your turn, and there are 5 matchsticks left, then you lose: go ahead and try it. (If you take 1, I take 3, and there is 1 left, so you lose. If you take 2, I take 2, so 1 left again and you lose again. If you take 3, I take 1, same old story. No matter what you do, I can make a choice that leaves 1 left on your next turn.) Similarly, we can reduce 9 to 5, 13 to 9, etc. What’s the pattern? If it’s your turn, and the number of matchsticks left is equivalent to 1, modulo 4, (like 5, 9, 13, ...) then I have a winning strategy no matter what you do. I should always choose my move to leave a number of matchsticks of the form $4k + 1$ for some $k$. That’s why, in the above example, I began by choosing 2 to leave you with 13 ($= 4 \cdot 3 + 1$) sticks; when you took 3 and left 10, I took 1 to leave 9 ($= 4 \cdot 2 + 1$). And so on; no matter how many sticks you chose I could always find the right number in my next step that left you again with $4k + 1$ for some $k$.

**Exercise**

Turn the above discussion into a proof (by induction on $n$, the number of initial matchsticks) that the first player can force a win, no matter what the initial number of matchsticks is, provided that $n \mod 4$ is not 1.
How to win at games

Nim is special, in that there’s a quick and easy calculation that tells me how to guarantee a win, just by looking at the current state. 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? The obvious answer is to use your computational resources to explore possible future states! The basic idea is that the first player should try to figure out his best possible outcome, assuming that after each of his moves the opponent tries to do his or her best also.

Given the zero-sum nature of the games we deal with, it is natural to associate a “score” or “outcome” value with terminal game states, and for the first player to take scores literally and the second player to negate the sign of the score. We’ll only deal with games in which scores are integers, although it would be quite straightforward to generalize and adapt the program that we develop to incorporate more general games. But the ideas behind game solving algorithms are already subtle and interesting enough in this limited context. With integer scores and this player-opponent +/- symmetry, the “do my best, assuming the opponent tries to do best” strategy can be explained very nicely using \texttt{Int.max} and \texttt{Int.min}: I try to maximize my (positive) outcome, while the opponent tries to minimize his/her (negative) outcome.

Game trees

We can explain the relevant game-theoretic concepts, by building a game tree whose nodes represent game states and whose edges represent game moves; each layer in the tree is labeled with the name of the player whose turn it is in all the states at that level. We call the players \texttt{Me} and \texttt{You}, or \texttt{Player} and \texttt{Opponent}, or \texttt{Maxie} and \texttt{Minnie} (or \texttt{MaxiMe} and \texttt{MiniMe}) if we want to emphasize the rôle of maximization and minimization in the way the two players decide on moves and strategy. The names we give to players aren’t important, as long as we keep clear which one goes first. The ML code implementing this algorithm won’t actually build trees like this, but instead uses recursion. But we can explain how the code runs by appealing to game trees and talking about traversing these trees and propagating information about possible outcomes.

The notion of a game tree makes sense for arbitrary 2-person games. We illustrate the ideas using the game Nim as a running example. For
obvious reasons, starting with a large number of sticks will require us to draw large tree diagrams. To avoid the tedium of type-setting large diagrams in verbatim mode, we'll focus on some smaller ones.

A Nim game tree, starting from an initial state with 3 matchsticks and MaxiMe to go first, MiniMe to go second, looks like:

```
  3  MaxiMe
 /   \
0 1 2  MiniMe
 | / \|
0 0 1  MaxiMe
 |   |
0   MiniMe
```

The nodes are Nim states, and each downward edge represents the effect of making a Nim move from a state, which takes us to a new state with fewer matchsticks; the difference between the source and target tells us how many matchsticks got removed in that move. Each level in the tree has a side-label saying whose turn it is in all game states at that level.

We want to assign to each state an outcome or value, which tells you who (eventually) wins from that state. We use the value 1 to indicate a win for MaxiMe, and −1 to indicate a win for MiniMe.

- First, we label the leaves. Leaf nodes for Nim have 0 matchsticks. If there are 0 matchsticks left, and it’s my turn, then you took the last one, so I won. So we label a leaf in a MaxiMe level with 1, and a leaf in a MiniMe level with −1. We’ll leave ? in places where we don’t yet have enough information. Labelling like this produces:

```
(3,?)  MaxiMe
 /     \
(0,-1) (1,?) (2,?)  MiniMe
 |     /  \
(0,1) (0,1) (1,?) MaxiMe
 |       |
(0,-1) MiniMe
```

So far, only the terminal states have definite outcome labels, of course!
• Next, propagate information up from nodes with definite labels to nodes labelled "?" when possible: We can do this at any "?" node whose children have proper labels. If it’s MaxiMe’s turn at a "?" node, we should label it with the maximum of the labels of its children, because MaxiMe will choose a move that leads to a node with the largest label; if it’s MiniMe’s turn at a node, we take the minimum of the children’s labels. We do this because we’re assuming that MaxiMe always plays in his best interest, so he tries to maximize the outcome (striving for +1 in Nim); and MiniMe always plays in her best interest, so she tries to minimize her outcome (striving for −1 in Nim).

- First stage: we can propagate from the deepest leaf to its parent node:

```
(3,?) MaxiMe
 / |  \
(0,-1) (1,?) (2,?) MiniMe
 | / |  \
(0,1) (0,1) (1,-1) MaxiMe
 |  |
(0,-1) MiniMe
```

- Next stage: we propagate from the next lowest leaf node, and to the node labelled (2,?). (We use parallelism here!) This gives

```
(3,?) MaxiMe
 / |  \
(0,-1) (1,1) (2,-1) MiniMe
 | / |  \
(0,1) (0,1) (1,-1) MaxiMe
 |  |
(0,-1) MiniMe
```

After this step we see in the rightmost child tree that if there are 2 matchsticks left, MiniMe should take 1, rather than 2: the first choice leads to a state (1, −1) whose label says that MiniMe wins, whereas the other choice leads to a state in which MaxiMe wins. We propagated the −1 label up to the node (2, −1) to record this knowledge.
Now we have labelled all three children of the root node; we can propagate once more, and end with:

```
    (3,1)          MaxiMe
   /       \             
(0,-1)  (1,1)  (2,-1)  MiniMe
  |       \        \       |
(0,1)  (0,1)  (1,-1) MaxiMe
   |                 |
(0,-1) MiniMe
```

The information at the root now says that when starting with 3 matchsticks MaxiMe should take 2, leaving 1, rather than taking 3 or 1.

- Each state in the game tree now has a label that indicates who has a winning strategy from that node. A smart MaxiMe can easily figure out from the structure of this tree how to ensure that no matter what MiniMe does, he will eventually win.

Exercise: draw and label the Nim game tree starting from 4 matchsticks. Then do it for 5 matchsticks.

What the minimax algorithm (as illustrated above) computes is the value of a game state, assuming both players will play optimally. It does not account for imponderable things like “if I do this, the chess board will look more confusing, so I think you’re likely to make a mistake”, or the fact that human players may resort to “psych” moves that fly in the face of logic. Intuitively, for a state in which MaxiMe goes next, the algorithm figures out what moves are possible, and for each move (recursively) finds the best value that MiniMe could obtain if she started from the state reached by MaxiMe’s move, and then takes the maximum. Symmetrically, for a state where MiniMe is to move next the algorithm computes the minimum value that MaxiMe could obtain, over all states reachable by a MiniMe-move.

Of course, in a game with a big search space, we cannot feasibly draw out the whole game tree, and the number of recursive calls needed to calculate optimal moves may be ridiculously large. Nevertheless we will first implement a full minimax algorithm, since this will be a useful exercise in functional programming. Later we will introduce ways to use bounds on how deeply to
explore the game tree, and heuristic functions to make estimates of the value of a game state when the state is too deep in the game tree. For Nim, as we’ve seen, there’s a perfect heuristic: is the number of matchsticks congruent to 1 modulo 4? For chess, a useful heuristic would take account of 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 minimax/game tree searching 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 Nim, Chomp, 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. Each of these players uses a strategy that 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.

So far, we’ve mainly used signatures for basic data structures, which you might perhaps expect to be in a library somewhere. (Indeed, the ML implementation does have lots of library signatures and structures.) But it’s unlikely that your library will come with a signature for 2-player, deterministic, perfect-information, zero-sum games.
3 Games

Let’s start with the signature for a game. As usual we assume that we have a structure `Seq:SEQ` implementing sequences.

signature GAME =
  sig
    exception Fail of string
    type state (* states of the game *)
    type move (* moves of the game *)
    val score : state -> int
    val moves : state -> move Seq.seq
    val step : state * move -> state
      (* REQUIRES m is in moves(s) *)
      (* ENSURES step(s, m) evaluates to a game state *)
  end

An implementation of this signature, i.e. a game structure, must specify:

- A type `state` whose values represent game states.
- A type `move` whose values represent the actions (game moves) allowed in a given game state.
- A function `moves` that computes the sequence of allowed game moves from a given state.
- A function `score` that tells us the outcome of terminal game states.
- A function `step` that tells us the state resulting from making a move from a state. For simplicity, `step(s,m)` requires that `s` must be a state for which the move `m` is legal. This requirement will be satisfied at call sites in our code. Moreover, we won’t ever do move generation in a game that is already over.

We included some comments in the signature, and the intention is that these comments impose some constraints on programmers who implement these signatures. Of course, merely writing the comments won’t make magic happen, and implementors are free to ignore these comments without being subject to any penalty. However, the requires- and ensures- specs here are
stated because they provide guidance in the implementation of well-behaved code. Pay attention throughout the code development to see where these comments help.

### 4 Nim

Here’s an implementation of Nim.

We declare an exception `Fail` that can be raised with a string argument used to indicate an error message.

```ocaml
structure Nim : GAME =
struct
  exception Fail of string
  type state = int
    (* The number of matchsticks left *)

type move = int
    (* How many matchsticks to pick up *)

  fun score 0 = 1 | score _ = raise Fail "Not a terminal state"

  fun moves pile =
    let
      val n = Int.min(pile, 3)
    in
      Seq.tabulate (fn x => x+1) n
    end;
    (* In a Nim state with k sticks left, you can pick up between 1 and min(k, 3) sticks *)

  fun step (pile, m) =
    if (pile >= m) then (pile - m)
    else raise Fail "Tried to make an illegal move"
    (* A move that takes m sticks is legal if m <= number of sticks. Doing this takes us to state with pile-m sticks. *)
end
```

10
• A state is represented by an integer (number of matchsticks left). A move is represented by an integer (which must be 1, 2, or 3).

• We could instead have defined the types `state` and `move` to be datatypes with constructors that don’t appear in the signature. What would have been the benefits of doing so?

• For `score`: if there are no matchsticks left, the game is over, and whoever’s turn it is next wins, because whoever took the last one has just lost. We define `score(0) = 1`, and remember that the second player will negate this.

• For `moves`: if there are fewer than three matchsticks, you can only take up to that many; otherwise you can take 1, 2, or 3. Note that `moves(p)` is designed to contain only legal moves, according to the Nim rules. In particular, when `p` is positive, 0 does not belong to `moves(p)`.

• For `step`: if you take matchsticks away that produces the obvious state; if you try to take away too many, disobeying the rules of Nim, an exception is raised.

How does this ML structure called `Nim` reflect our earlier discussion of Nim game trees? Here again is the Nim game tree whose root is the state with 3 matchsticks, MaxiMe to go next, nodes annotated with the “values” that we computed by propagation using min/max steps:

```
  (3,1) MaxiMe
   /     \
 (0,-1) (1,1) (2,-1) MiniMe
  |      / \
 (0,1)  (0,1) (1,-1) MaxiMe
   |    /
  (0,-1) MiniMe
```

Using the functions defined in the Nim structure, each of the following facts has a pictorial echo in the above drawing. We use the math notation for sequences here, so don’t expect ML to reproduce the same results literally!
When we implement the minimax algorithm using recursive functions, it will turn out that our implementation does calculate the same labels for states as the ones appearing in this picture.

5 Minimax

Let’s implement a generic player for games, who always uses the minimax algorithm to decide on the best move to take. Instead of building game trees, we define a pair of functions from states to scores (one for Player1, one for Player2) and use them to define a function from states to moves (a best move selection strategy for Player1, assuming Player2 always tries best). In figuring out the best move for the player whose turn it is, we make recursive calls to find the best moves for the other player from the possible next states. Thus we will need a pair of mutually recursive functions. ML supports mutual recursion very elegantly. The recursive flow of control mimics the bottom-up propagation of labels in our previous discussion of the ideas behind minimax.

Since the idea of minimax makes sense for any game, not just for Nim, we encapsulate this algorithm by means of a functor (Figure 1).

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. (Non-terminal states always have at least one move out of them.) In cases like this there’s no need to supply a “zero” element to the reduce operation. So we’ll simply assume that our signature `SEQ` for sequences actually includes a function

```
reduce1 : ('a * 'a -> 'a) -> 'a seq -> 'a
```

with the obvious specification: when `s` is non-empty, with first item `x` and “tail” `s'`, `reduce1 g s` behaves just like `reduce g x s'`. 
Note the opportunities for parallelism in implementing minimax: at each level, you can 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.)

In class, we developed estimates of the asymptotic work and span for the functions below.

To indicate mutually recursive definitions, you write and instead of fun for the second definition. In general you can declare two, three, ..., as many as you need, mutually recursive functions, using and to link the definitions, as in

\[
\begin{align*}
\text{fun } f_1(x_1) &= e_1 \\
\text{and } f_2(x_2) &= e_2 \\
\text{and } f_3(x_3) &= e_3
\end{align*}
\]

Let’s isolate the state evaluation functions for the two players.

\[
\begin{align*}
\text{fun } F(s) &= \\
&\text{let } \\
&\quad \text{val } M = \text{moves } s \\
&\text{in } \\
&\quad \text{if } (\text{null } M) \text{ then score } s \text{ else } \\
&\quad \\quad \text{reduce1 Int.max } (\text{map } (\text{fn } m => G(\text{step}(s, m))) M) \\
&\text{end} \\
\text{and } \\
\text{G}(s) &= \\
&\text{let } \\
&\quad \text{val } M = \text{moves } s \\
&\text{in } \\
&\quad \text{if } (\text{null } M) \text{ then } -(\text{score } s) \text{ else } \\
&\quad \\quad \text{reduce1 Int.min } (\text{map } (\text{fn } m => F(\text{step}(s, m))) M) \\
&\text{end}
\end{align*}
\]

We can use equational reasoning to talk about game behavior. In particular, it follows from the ML definitions that (when the game is Nim):

\[
\begin{align*}
F 3 &= \text{reduce1 max } < G 2, G 1, G 0 > \\
G 2 &= \text{reduce1 min } < F 1, F 0 >
\end{align*}
\]
functor MaxiMe(Game : GAME) : PLAYER =
struct
structure Game = Game
open Game

fun F p = let
  val M = moves p
  in
    if (null M) then score p else
      reduce1 Int.max (map (fn m => G(step(p,m))) M)
  end
and
  G p = let
    val M = moves p
    in
      if (null M) then ~(score p) else
        reduce1 Int.min (map (fn m => F(step(p,m))) M)
    end

type edge = move * int

fun maxedge ((m1,v1),(m2,v2)):edge = if v1 < v2 then (m2,v2) else (m1,v1)
fun maxbestedge L = reduce1 maxedge L

fun player p = let
  val M = moves p
  val (m,_) = maxbestedge(map (fn m => (m, G(step(p,m)))) M)
  in
    m
  end
end

Figure 1: MaxiMe
G 1 = reduce1 min < F 0 >
G 0 = ~1
F 1 = reduce1 max < G 0 > = ~1
F 0 = 1

So we see that
G 2 = reduce1 min < ~1, 1 > = ~1
G 1 = 1
F 3 = reduce1 max < ~1, 1, ~1 > = 1

Check that you see the relationship between these results and the shape of the labelled tree for Nim that we developed earlier. As we promised, the bottom-up propagation of labels gets implemented by recursive function calling. Here’s the labeling we showed earlier:

```
(3,1) MaxiMe
/  |  \   
(0,-1) (1,1) (2,-1) MiniMe
  /  |  
(0,1) (0,1) (1,-1) MaxiMe
    |  
(0,-1) MiniMe
```

Here is the game tree but with each state paired with the F or G value calculated for that state by the appropriate player:

```
(3,F 3) MaxiMe
/  |  \   
(0, G 0) (1, G 1) (2, G 2) MiniMe
  /  |  
(0, F 0) (0, F 0) (1, F 1) MaxiMe
    |  
(0, G 0) MiniMe
```

With the values calculated above, you can see that these trees are identical!
**Terminating games**

A game is terminating if there is no infinite sequence of legal moves from any state. We can capture this definition nicely by means of an ML function

\[
\text{duration : state -> int}
\]

that calculates the longest sequence of consecutive moves possible from a given state:

\[
\begin{align*}
\text{fun duration \ (s : state) \ : \ int =} \\
\text{let} \\
\text{val M = moves s} \\
\text{in} \\
\text{if (null M) then 0 else} \\
\text{1 + reduce1 Int.max} \\
\text{(map (fn m => duration(step(s,m))) M)} \\
\text{end}
\end{align*}
\]

**Definition**

A game is *terminating* iff for all \( s : \text{state} \), \( \text{duration}(s) \) terminates.

Since \( \text{duration} \) is a function of type \( \text{state -> int} \), saying that \( \text{duration}(s) \) terminates is the same as saying that it evaluates to a value of a type \( \text{int} \). And it is easy to see (and prove!) that this value is non-negative. Also, if \( \text{step}(s,m) = s' \) it follows that \( \text{duration}(s') < \text{duration}(s) \).

So for a terminating game, we may be able to use induction on the duration of states, to prove properties of the code we write. Of course we will need to make some (reasonable!) assumptions about the basic properties of the ingredients of a game implementation: we assume that the functions \( \text{moves}, \text{step} \) and \( \text{score} \) are total.

It’s easy to see that \text{Nim} is a terminating game: every legal move strictly decreases the number of sticks, and this number can never go negative.

Here is an example of how to use induction on game duration.

First, note this lemma about \( \text{Int.max}, \text{Int.min} \), and integer negation:

For all integers \( m \) and \( n \), \( \text{Int.max}(\neg m, \neg n) = \neg \text{Int.min}(m,n) \).

As an easy corollary, for all integer sequences \( v_1, \ldots, v_k \)
reduce1 Int.max < ~v1, ..., ~vk >
= ~(reduce1 Int.min < v1, ..., vk >).

Theorem
For a terminating game, the functions F and G as defined above are total functions, and for all s:state, (F s) = ~(G s).

Proof
By induction on duration(s).

• Base case: When duration(s) = 0, i.e. when moves(s) is empty. Then by definition of F and G we have

   (F s) = (score s)
   (G s) = ~(score s)

so the result holds in this case.

• Inductive step:
Let s be a state with duration(s) > 0 and assume as IH that for all states s' with smaller duration, (F s') = ~(G s'). We must show that (F s) = ~(G s).

Let moves(s) = ⟨m₁,...,m_k⟩ and s_i = step(s,m_i) for i = 1...k. Then k > 0 by assumption, and

   F(s) = reduce1 Int.max {G(s₁),...,G(s_k)}
   G(s) = reduce1 Int.min {F(s₁),...,F(s_k)}

But for each i, duration(s_i) < duration(s), so by IH F(s_i) = -G(s_i). Hence

   F(s) = reduce1 Int.max {G(s₁),...,G(s_k)}
   = reduce1 Int.max {−F(s₁),...,−F(s_k)}
   = −reduce1 Int.min {F(s₁),...,F(s_k)}
   = −G(s)

as required.
6 Self Test

1. Define a structure `GeneralNim` that implements the GAME signature and represents the “general Nim” two-person game, in which the state is a finite list of piles, each pile consisting of a number of matches; the player whose turn it is must take away any non-zero number of matches from one of the piles. The player who removes the last match loses the game. For example, in state `[3,4,5]` there are possible moves to the following states:

   
   [0,4,5], [1,4,5], [2,4,5],
   [3,0,5], [3,1,5], [3,2,5], [3,3,5],
   [3,4,0], [3,4,1], [3,4,2], [3,4,3], [3,4,4]

   The only terminal state is one in which all piles are 0.

2. Using the MaxiMe functor and your GeneralNim implementation, figure out the best move for Maxie from start state `[3,4,5]`.

3. What is the best possible outcome for Maxie from initial state `[15]`?

4. Let F and G be the functions defined as above, using the Nim game implementation. So for \( n > 2 \),

   \[
   F \ n = \text{reduce1 Int.max } <G(n-3), G(n-2), G(n-1)>
   
   G \ n = \text{reduce1 Int.min } <F(n-3), F(n-2), F(n-1)>
   \]

   and so on. Prove by induction on \( n \) that for all \( n > 0 \),

   \[
   F \ n = \text{if } (n \mod 4 = 1) \text{ then } \neg 1 \text{ else } 1
   \]