Project 3: Game

7:00 PM, Dec 5, 2019

Contents

> "There is no finite game unless the players freely choose to play it. No one can play who is forced to play. It is an invariable principle of all play, finite and infinite, that whoever plays, plays freely. Whoever *must* play, cannot *play*."
>
> —*Finite and Infinite Games*, James P. Carse

> "In chess too," he said at last, "there's a limit to the forecasts one can make. The best possible move, or the most probable one, is the one that leaves one's opponent in the least advantageous position. That's why one way of estimating the expediency of the next move consists simply in imagining that move has been made and then going on to analyze the game from your opponent's point of view. That means falling back on your own resources, but this time putting yourself in your enemy's shoes. From there, you conjecture another move and then immediately put yourself in the role of your opponent's opponent, in other words, yourself. And so on indefinitely, as far ahead as you can. By that, I mean that I know where I've got to, but I don't know how far he's got."
>
> —Muñoz, *The Flanders Panel*, Arturo Pérez-Reverte

1 Introduction

For your final project, you will implement a two-player, sequential, finite-action, deterministic, zero-sum game of perfect information. Let's define what that means:

- A **two-player** game has two players. Tic-tac-toe is a two-player game; Hearts is not.
- A **sequential** game is one in which only one player moves at a time. Monopoly is a sequential game; *Rochambeau* (i.e., rock-paper-scissors) is not.
- A **finite-action** game is one in which there is a finite number of legal moves available to a player when it is his or her turn to move. Battleship is a finite-action game; soccer is not.
- A **deterministic** game is one that does not depend at all on chance. Its progress is entirely a function of selected moves. Checkers is deterministic; backgammon is not.
- A **zero-sum game** is one in which what is good for one player is equally bad for the other, and vice versa. All the examples in this section are zero-sum games.
- A game of **perfect information** is one in which both players witness the entire progression of the game. Chess is a game of perfect information; poker is not.

In addition to implementing a game, you will also write a general purpose artificially intelligent (AI) player that can play any game of this sort. Your AI player will play a game using the minimax algorithm, together with a helper procedure that estimates the values of intermediate game states.

In lab, you have (or will have) seen two simple games: GoFirst and Nim. This project entails implementing a more complex game, with a more sophisticated `estimateValue` procedure and AI player.

The large-scale structure of the completed project is this: There's a "Referee" module, which incorporates a "Game" module and two "Player" modules. The main procedure in the referee module creates a game (using the Game module), and then alternately asks the two players to make moves in that game, keeping track of whether one of them wins.

Each "Player" module describes either a human player or an AI player. Depending on how you construct the "Referee" module, it can run a game between two humans, or between a human and an AI, or between two AI players. The human player is actually just a user interface that asks a real human which move to make. The AI player selects moves based on a generic strategy called *game-tree search*. By "generic strategy", we mean that the approach does not depend on which game is being played. Of course, the strategy must ultimately use something specific to the game, and that specific thing is called a *static game-state evaluator*: it is a procedure that assigns a number to each game state (called the *value* of the state) that reflects how good/bad that state is for the AI player. In our implemention, the state evaluator is called `estimateValue`. (The reason for the modifier *static* will be made clear later.)

2 Pick A Game!

To start this assignment, you must pick a game.

We suggest implementing one of the following "official" games:

- Connect 4
- Mancala

We describe our version of these games at the end of this handout. If you choose to implement one of these official games, please be sure to read our description of that game carefully, because our rules may differ slightly from the rules you know.

If you choose one of these official games, we guarantee that at least one of the TAs will be expert enough in your game to be a good source of advice and guidance to you. Still, you may choose to implement something else, like checkers; but if you do, we cannot promise you that any of the TAs will be experienced enough with your particular game to help you extensively. If you want to implement an "unofficial" game, please seek approval from the TAs first.

Traditionally these games are played on a board of fixed size. But, in the spirit of abstraction and elegance, we are requiring that your implementation handle boards of varying sizes (e.g., larger boards, or more pieces in play, etc.).

3 Assignment

Once you've picked a game, there are three parts to this project.

1. You have to implement your game. This involves writing a module with data types capable of keeping track of the state of the game, and a few procedures that provide information and generate new states.
2. You have to complete the implementation of the human player. We've provided part of the code (which handles receiving input), but you need to handle the rest.
3. You have to build an AI player capable of playing your game. AI stands for artificial intelligence. Your AI player should play intelligently, where "intelligent" is understood to mean rational.

3.1 Setup

Source code can be downloaded from the course website. It should include the following 9 files:

- `README`
- `GameName.re` (replace this with the name of your game - Connect4.re or Mancala.re)
- `HumanPlayer.re`
- `AIPlayer.re`
- `Referee.re`
- `SigGame.re`
- `SigPlayer.re`
- `CS17SetupGame.re`
- `ReadLineSyncTest.re` (not used for the project, only used to test the package we are about to install)

Once you initialize a project and download the files into the src folder of your project, you need to download a package to allow your user to read in input. To do this, navigate in terminal to the project directory and type `npm install readline-sync`. This will install a package to allow your game to run a human player. To test that this was installed correctly, you can compile and run `ReadLineSyncTest.bs.js`. If you can type a sentence and hit enter and get that sentence repeated back to you, then it installed correctly. Note that you won't be able to compile your project right upon downloading, because your game will not have all of the required procedures to implement the `Game` module and will throw an error. Thus, to make it compile, add failwith stubs for all required procedures.

Another note on the source code: Based on your implementation, you may require "open"-ing of different files than we have in the source code. Thus, while you should **not** alter SigGame.re, SigPlayer.re, or CS17SetupGame.re, you may need to change the headers of other files.

3.2 Game Signature and Module

For the first part of this project, you will need to understand a game abstract data type. This abstract data type captures (much of) what is common to two-person, sequential, finite-action, deterministic, zero-sum games of perfect information.

Note: The content of this section is reminiscent of (if not copied verbatim from) Lab 11.

Game Signature Here is the `Game` signature:

```
module type Game = {
  /* TYPES */
  type whichPlayer = P1 | P2;
  type status =
```

```
    | Win(whichPlayer) /* this says who wins */
    | Draw
    | Ongoing(whichPlayer); /* this says whose turn it is */
  type state;
  type move;

  /* INITIAL GAME STATE */
  let initialState: state;

  /* TYPE CONVERSIONS */
  let stringOfPlayer: whichPlayer => string;
  let stringOfState: state => string;
  let stringOfMove: move => string;
  let moveOfString: string => move;

  /* GAME LOGIC */
  let legalMoves: state => list(move);
  let gameStatus: state => status;
  let nextState: (state, move) => state;
  let estimateValue: state => float;
};
```

This signature has five main components:

- Types:
  - There is a variant type `whichPlayer` that specifies one of the two players.
  - There is a type `status` that summarizes the status of the game (e.g. whose turn it is if the game is not over).
  - One type that you must define for your particular game is `state`, which is used for representing everything about the progress of the game. How is the current board represented, and whose turn is it? For example, for Checkers, this would involve specifying for each square of the board whether the square was occupied by red or black or neither.
  - Another type that you must define for your game is `move`, which is used for representing all the possible moves a player might ever be able to make. For example, a move in Nim would specify how many matches to take. A move in Checkers might specify the location of the piece to move and the location to which to move it. Note that a particular value of type `move` might not be legal in a particular game state. For example, for Tic-tac-toe, a move might be an integer from 1 to 9 indicating which square is to be marked. Note that the move `1` is still a move even in the case where the upper left square of the board has an "X" in it – it's just not a *legal* move!
- A non-procedure value:
  - `initialState`: The initial state of the game, i.e. the state just before the first move.
- Type conversion procedures: How does one go from internal representations to an external representation as a string (and vice versa, in the case of `move`)?

  - `stringOfPlayer`: Produces the string representation of a player.
  - `stringOfState`: Produces a string representation of a game state: what information do both players need to know in a turn? For tic-tac-toe, this might produce a string that when printed actually shows a 3×3 board with lines dividing up the squares, etc., and a note at the bottom indicating whose turn it is.
  - `stringOfMove`: Produces a string representation of a move.
  - `moveOfString`: Produces an internal representation of a move, given a string. This involves parsing a string and failing if the input is not valid.

- Game logic: These are the operations that define your game.
  - `legalMoves`: Produces a list of all the legal moves at a given state.
  - `gameStatus`: Produces the status of the game at a given state.
  - `nextState`: Given a state and a move, produces the state that results from making that move.

- A static game-state evaluator
  - `estimateValue`: provides an estimate of the value, to player 1, of a particular state in the game. This value should be positive for states where player 1 seems to be winning (and especially for those in which player 1 has actually won!), negative for those in which player 1 seems to be losing (and especially for those in which player 1 has actually lost), and have intermediate values for game-states in which the advantage to player 1 is not so clear. A very simple `estimateValue` assigns the value 1.0 to each winning state for player 1, -1.0 to each losing state for player 1, and 0.0 for all other states. Your actual `estimateValue` should be more discriminating than this. Also: the values need not be between -1.0 and 1.0: all that matters is that larger values mean that a situation is better for player 1. Having negative values for situations that are good for player 2 makes it easier to see how the game is going, but is not essential. The name `estimateValue` arises because of the way that this particular procedure gets used by the minimax algorithm.

Game Module Your next task is to write a module that implements this `Game` signature for your chosen game. Here's a road map for how you might proceed:

1. Write a template for a module that implements a testable version of your game, starting like this:

```
module Connect4 = {

  type whichPlayer = P1 | P2;
  type status = Win(whichPlayer) | Draw | Ongoing(whichPlayer);

  type state = ...

  type move = ...
```

```
  let initialState = ...
  ...
};
```

2. Fill in the types for `state` and `move`. We have filled in the `whichPlayer` and `status` types for you.

3. Add **stubs** for all of the procedures in the `Game` signature. A stub is just a procedure header plus a body which does nothing useful, or reports an error. For example, here's a stub for the `legalMoves` procedure:

```
let legalMoves: state => list(move) = fun
  | _ => failwith("not implemented");
```

4. The `Connect4` module is concrete. You should leave it concrete so that your helpers and types and values are fully visible, making it easier to test them.

   However, you should ensure that your Game module (e.g. `Connect4`) implements the `Game` signature. To do this, use the `open` directive to read in the `Game` signature (in `SigGame.re`). Finally, try making an abstract module from your game module using signature ascription, e.g.

```
module MyGame : Game = Connect4 ;
```

   or, equivalently,

```
module MyGame = (Connect4 : Game) ;
```

   If this succeeds, your module is consistent with the `Game` signature. If not, the error message should give you a hint as to what is missing or wrong.

5. Gradually fill in all of the stubs you just wrote until they're working, *except* `estimateValue`.

At this point, you're almost ready for two human players to play your game.

(See Section 4 for instructions.)

3.3 Player Signature and Modules

The second major part of this project is to implement an AI player that can intelligently play any game that implements the `Game` signature. To do this, you will implement the minimax search algorithm discussed in lecture, which looks ahead several turns and selects the move that appears to be most beneficial in the long run. But first, let's do something simpler: understand the `Player` signature, and create a `HumanPlayer` that conforms to this signature.

Here's the signature for a `Player`:

```
module type Player = {
  module PlayerGame: Game;
  /* given a state, chooses a move */
  let nextMove: PlayerGame.state => PlayerGame.move;
};
```

A player's only task is to figure out its next move, given the state of the game.

As you can see in the first line, a player has a submodule that satisfies the `Game` signature. Why does it make sense for a player to have this? A player needs to know what game it is playing — this is the submodule.

In `HumanPlayer.re`, we partially provide a `HumanPlayer`.[1]

Your first player-related job is to complete the `HumanPlayer` implementation by looking at the TODO.

3.3.1 Parsing Human Input

There's one problem you'll encounter here (in `moveOfString`): what do you do when you ask the user to type a move (e.g., a number from 1 to 3) and the user types "andrew" rather than "1", "2", or "3"? Well, you want to ask the user to retype the move, but perhaps your `moveOfString` procedure has already tried to convert "andrew" to a number between 1 and 3, been unsuccessful, and hit a "failwith" case, or maybe `string_of_int` has failed by "raising an exception." At this point, your program has halted, and there's nothing you can do.

We addressed this with something called a "try" expression. Here's an example. Let's start with a simple procedure:

```
let f = fun
  | x => string_of_int(1000 / x);
```

The procedure can be applied to arguments in the usual way:

```
f(25);
f(0);
```

The first invocation is successful:

```
let f : int => string = <fun>;
- : string = "40"
```

The second is not:

```
Exception: Division_by_zero.
```

As you can see, when Reason evaluated $f(0)$, there was a divide-by-zero exception raised (i.e., there was a problem, and something in the Reason interpreter noticed it, and did something called "raising an exception", which ended up halting the program and printing a message).

The structure of a try expression is

```
try (<exp>) {
  E1 => ...
  E2 => ...
}
```

[1] Not literally, of course! We provide an implementation of `Player` that determines its next move based on user (i.e., human) input.

where `<exp>` is some expression; if it evaluates without exceptions to some value v, then the value of the try-with expression is just v. On the other hand, if there's an exception, we can "pattern-match" the exception to produce a different value (and to prevent program termination). Here's a modified version of our procedure, and the new results:

```
let f = fun
  | x =>
    try (string_of_int(1000 / x)) {
    | _ => "Divide by zero"
    };

f(25);
f(0);
```

```
let f : int => string = <fun>;
- : string = "40"
- : string = "Divide by zero"
```

The code "caught" the divide-by-zero exception and provided a replacement value.

We provided a try-catch block in HumanPlayer.re to account for invalid moves, like typing in "k" instead of an integer.

The `nextMove` procedure of a `HumanPlayer` module only handles `Failures`, not `Exceptions`. This is partly to enforce that we'd like you to fail with your own custom error messages for a user. Also, this way, actual `Exceptions` still fail – for instance, pressing CTRL-D (control D) while reading input raises an `Exception`, and we'd like to make sure the program still fails then, so you can safely exit the game even when asked to enter a move.

Another potential concern when implementing `moveOfString` is *parsing*. When a move consists of multiple pieces (like the (x, y)-coordinates of a token on a checkers board), the best way for a human user to represent the move is not clear.

We recommend that if your game has complex moves, you try separating them with a comma. Let's say you're implementing checkers, and you want a move to consist of four numbers (the first two numbers being the row and column of the token to be moved and the last two being the row and column in which to place the token). Then, examples of parse-friendly input might be `1,2,2,3` or `2,0,1,0`.

Note that this is easier to parse than it would be if you included parentheses, like (1,2,2,3).

The substring `(sub)` and `index` procedures in the `String` module may be especially helpful in getting the first number, second number, etc. from a string.

Note: If you want to participate in the in-class tournament to see how your AI stacks up against other students', makesure the "move" your user inputs is just a single integer.

3.4 The AI Player

Now we come to the interesting part: writing code that will play the game effectively.

Here is the syntax for creating an `AI` module of type `Player`:

```
module AIPlayer = (Game: Game) => {
  module PlayerGame = ...
  ...
};
```

Again, you'll likely want to ensure that your `AIPlayer` module implements the `PLAYER` signature; to do this, you'll want to do something similar to what we did for the `GAME` signature and Game module.

As noted above, you only have to implement one procedure, `nextMove`. But implementing this procedure is not necessarily straightforward. Here's a road map for how you might proceed:

1. In the `AIPlayer` module, replace the stub for `nextMove` with a procedure that simply gets the set of legal moves and returns any one of them. Verify that you can play a game against your AI player, and that everything works fine (except that the AI player makes naive moves).

2. Make a first improvement of `estimateValue`, having it return the max value if P1 wins, the min value if P2 wins, or 0 otherwise.

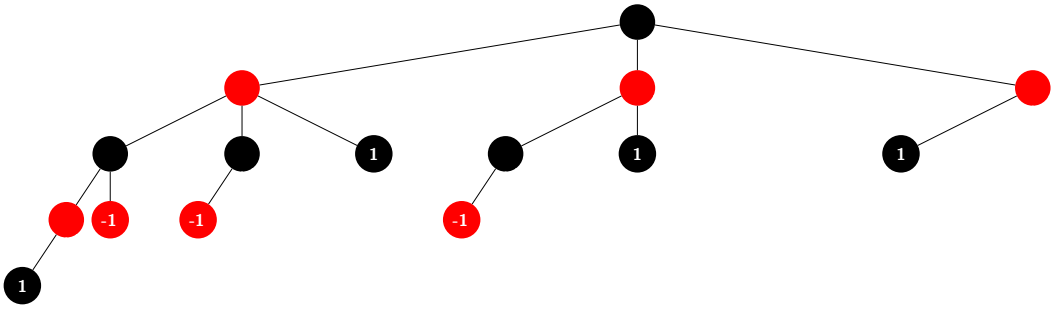
3. Rewrite `estimateValue` to do something sensible. The specifics of this procedure will depend on the game that you choose. This is the hardest part of the assignment for most games, and the one that defines how good your AI is.

4. Write a new version of `nextMove` which first uses `estimateValue` to evaluate the results of each possible move, and then plays the best one. Try out this new version—it should be a little smarter.

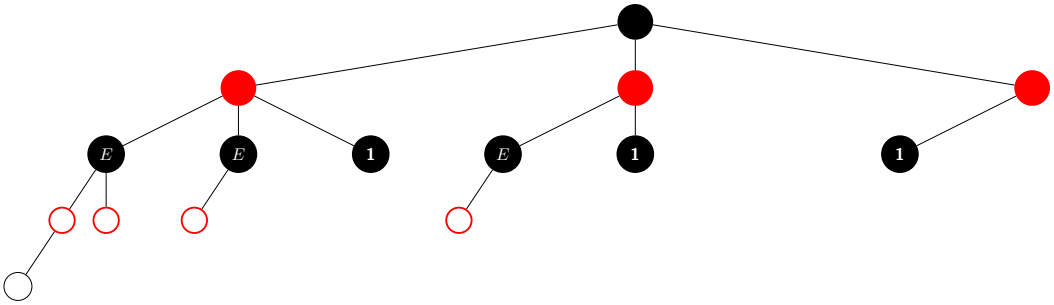
5. Implement limited-depth minimax search. You should look ahead something like 3 to 5 levels. (If your game has many possible moves, deep lookahead will be infeasible. If it has only two or three at each state, then quite deep lookahead may be feasible, as long as `estimateValue` is reasonably efficient.) Typically, extra levels of look ahead are worth much more than an improved `estimateValue` procedure, as long as `estimateValue` is at all reasonable.

   You can imagine a huge tree, where each node is a possible state of the game, and each edge leaving a node corresponds to a move by one player or the other. A leaf in this "game tree" is a state in which the game is over, either because player 1 or player 2 has won, or because it's a draw. When the AI player has to make a move, one way to decide on a move is to look at all possible future states of the game (i.e., the subtree rooted at the current state) and see which move leads to the best possible outcome.

   But looking at that whole subtree may not be feasible. Instead, if we look only at the current state, its children, and its grandchildren, we have a depth-3 tree sort of sitting "inside" the game tree. We could look at this smaller tree, and try to decide how to move based on it. To do so, we try to pick a move with the property that when the opponent picks their best move, we still end up in the best possible position. (This is the one-level version of minimax.) For this to work, we need to evaluate how much we like a particular "grandchild" state, i.e., we need to *evaluate* a *leaf* of our tiny tree to see how good a state it is. For that, we use the `estimateValue` procedure in the game.



Above is the game subtree for a NIM game in which there are four pieces left, and it is your turn. The leaves store a value corresponding to the status of the game: 1 if player 1 wins, -1 if player 1 loses.



The bolded nodes constitute the tree that your minimax algorithm will consider, i.e. not necessarily the whole tree. E is an estimate for the value of the game given that state.

Note: The AI player can only interact with the game module through the `GAME` signature. It's not allowed to access any internal representation of states or moves or anything else. So for an AI player to decide on a move, it must ask the game what are the possible moves, and the resulting next states, and then the moves from *those* states, and so on. It must evaluate the resulting states to see which ones it likes or dislikes, and move accordingly.

3.5 How Everything Fits Together

In our support code, we have provided you with eight files: `SigGame.re`, `SigPlayer.re`, `Game.re`, `HumanPlayer.re`, `AIPlayer.re`, `Referee.re`, as well as the setup file and an empty `README` which you should fill out. The first two code files provide the `Game` and `Player` module signatures. They contain all of the types and function type signatures that you will implement in `Game.re`, `HumanPlayer.re`, and `AIPlayer.re`. We'll now walk through how the modules fit together in the other four files.

`Game.re` contains the skeleton for a concrete implementation of the `Game` signature, which we have called `Game`, but which you should rename. If you're writing the game Connect4, we'd recommend a name like `Connect4`! All of the details of your game implementation will reside in this concrete module. You will also write test cases in this file, right after the module definition.

In `HumanPlayer.re` and `AIPlayer.re`, we use the same idea to create two concrete implementations of `Player`, which we call `HumanPlayer` and `AIPlayer`, respectively. Notice that in `SigPlayer.re`, we specify that `Players` contain a submodule called `PlayerGame` which needs to be of type `Game`. Thus, both `HumanPlayer` and `AIPlayer` must contain a `Game`, which is demonstrated in the following lines:

```
module HumanPlayer = (MyGame: Game) => {
  module PlayerGame = MyGame;
```

(Note that the first few lines for `AIPlayer` are nearly identical.)

You'll notice that `HumanPlayer` is a `functor`. A `functor` takes in a module and uses it to produce another module. In this case, the module the `HumanPlayer` takes in is the `MyGame` module (of type `Game`), representing the game that it is playing.

This means that, to create a `HumanPlayer` for a game of `Connect4`, you'd have to write something like

```
module TestGame = Connect4;
module HumanConnect4Player = HumanPlayer(Connect4);
```

Notice that you can't use `let` to create a new Player; you have to use the "module" keyword to create one.

The `module PlayerGame = MyGame` line inside the module definition defines the `PlayerGame` submodule to be the `Game` that you implemented in `Game.re`.

At this point, to reference any `Game` procedures here, you'll need to use the *qualified name*, i.e. the name must be preceded with `PlayerGame` and a dot. For example, to refer to `legalMoves`, you'll have to write `PlayerGame.legalMoves`. (If you get tired of using qualified names, you can add the command `open PlayerGame` and then use the unqualified names.)

Finally, we have `Referee.re`, which will conduct the actual game play.

4 Let's Play!

As we said earlier, the `Referee` module is in charge of running games. The referee is a functor. It takes in a Game and two Players that both play the given Game.

```
  module Referee =
         (
           MyGame: Game,
           Player1: Player with module PlayerGame = MyGame,
           Player2: Player with module PlayerGame = MyGame,
         ) =>
  ...
```

To let two HumanPlayers play one another at Connect4, you'd write something like:

```
module Ref = Referee(Connect4, (HumanPlayer(Connect4)), (HumanPlayer(
    Connect4)));
```

Think about what you would write instead if you want to pit a HumanPlayer against an AIPlayer, an AIPlayer against a HumanPlayer, or two AIPlayers against each other.

To start a game, invoke the `playGame` procedure in the resulting module, i.e.

```
Ref.playGame();
```

Note that the argument to `playGame` is of type `unit`, i.e. it is a zero-element tuple.

To do this, bring your code into the environment by using `open` directives. Next, construct a module by applying the functors. Finally, invoke the `playGame` procedure.

```
open Game;
open HumanPlayer;
open Referee;

module Ref = Referee(Connect4, (HumanPlayer(Connect4)), (HumanPlayer(
    Connect4))); /* change as necessary */

Ref.playGame();
```

You can do something similar if you want to get an AI player involved. Just remember that you have to bring your AI player functor into the environment using a `open` directive.

5 Getting Started

Here's an overview of the major steps that need to be taken in order to complete the project. This is one possible path you could follow, but feel free to complete the project in whichever order works best for you and your partner. For more specific information, consult the rest of the PDF, Piazza, or your friendly TA staff!

- You should start by defining the attributes and rules of your game in `GameName.re`. This includes things like defining your board, valid states, and legal moves. Ensure that all the methods and abstract data types in `GameName.re` all implement the signatures in `SigGame.re`. **Note that you should not modify `SigGame.re`**. You should also rename `GameName.re` to be either `Connect4.re` or `Mancala.re`.
- Next steps may be to fill in the `HumanPlayer.re` and `AIPlayer.re` files, which should contain methods and types related to the player, human or artificial. The bulk of the artificial intelligence infrastructure for your game should be in `AIPlayer.re`, minimax in particular.
- Test your game, not only in the traditional "checkExpect" fashion of CS17, but also ensure that the game behaves as expected in all four Game modes (Human vs. Human, Human vs. AI, AI vs. Human, AI vs. AI). If you can't beat your AI, that's a good sign! Note that you should test `HumanPlayer.re`. One way to approach this is by creating situations in which the next move is obvious, and making sure that your AI picks the correct move.

6 Notes

- There are some constructs that Reason will let you use but that we, the CS 17 instructors, will not. Specifically, you are not allowed to use mutation. (If you don't know what this means, don't worry; you won't use it by accident.) As always, if you are unsure about whether a certain Reason construct is allowed, ask the TAs.

- Please don't "hard code constants" (i.e., have them appear all over your program). Instead, define them at the top of your module. For example, instead of using the actual integers 5 and 7 in the Connect Four `initialState` to describe the number of rows and columns in the game board, use an identifier called `initialRows`, and another called `initialCols` and assign them the values 5 and 7. To do this, you should start your Connect4 module with

```
let initialRows = 5;
let initialCols = 7;
```

  and then your initialState should be built using "initialRows" and "initialCols". Doing so makes it easy to change the board dimensions later.

- The Reason `List` module has a number of procedures that should come in handy. Documentation can be found online at:
  `https://reasonml.github.io/api/List.html`.

  Documentation for the Reason Pervasives module (i.e., the procedures that are available in any Reason program) can also be found online at:
  `https://reasonml.github.io/api/Pervasives.html`.

  Old homeworks, labs, and lecture notes may also contain useful procedures.

- Here are some things you should know about strings:

  - The infix operator `++` concatenates two strings.
  - In a string, `\n` (backslash n) represents a newline character. For example, if the string

```
"hello\nworld"
```

    is printed out,[2] it will have a line break in between "hello" and "world".
  - `string_of_int` and `string_of_float` return a string representation of the number passed to them.
  - `int_of_string` and `float_of_string` return a number based on the string passed to them.
  - `Js.String.split(delimiter, str))` returns a procedure that breaks a string into a list of strings; if you implement a game whose moves are multiple numbers, you might want to first do this and then call `int_of_string` on the elements of the list.

    If you'd like to use the Js.String module, put this line at the beginning of your file:
    `open Js.String;`
    If you want to learn more about the `Js.String` library module, check out:
    `https://bucklescript.github.io/bucklescript/api/Js.String.html`

[2] Strings will include these special characters just as you typed them; to print them out, use the print_string operation. Unlike all the procedures we have written this semester, this operation does not only return a value; instead, it does something—namely, it prints out a message. We wrote the HumanPlayer for you because it uses operations like this one that aren't purely functional (because it has side-effects other than the value it returns; in this case the procedure returns a unit, `()`, an empty value in Reason).

- `Js.Re.exec_([%re "(regexp here)"], str))` returns an `option(Js.Re.result)` that contains a list of strings matched by your regular expression. Regular expressions are tools for complex pattern matching on strings, which may be helpful for parsing input.

  If you'd like to use the Js.Re module, put this line at the beginning of your file: `open Js.Re;`
  If you want to learn more about the `Js.Re` library module, check out:
  `https://bucklescript.github.io/bucklescript/api/Js.Re.html`

7 Handing In

7.1 Design Check

Design checks will be held on Nov 19-21, 2019. We will send out an email detailing how to sign up for design checks, so please check your inbox periodically, and sign up as soon as possible. You must decide which game you will implement by the design check.

Reminder: You are required to pair program this project. We recommend finding a partner as soon as possible, as you will not be able to sign up for a design check until you have one.

There are three components to the design check.

First, you should have at least a partial implementation of the Game signature. At minimum, you should fill out the type definitions for `move` and `state`, as well as writing the `initialState` value.

In addition, what you have should be well written, organized, and commented. You should be able to explain everything you've written to your TA.

Second, you should be able to describe how you intend to implement minimax. Specifically, you should be able to answer the following questions:

- Which procedures in your `Game` module will the AI player use? Why?
- What should the AI player do if there are no legal moves available? If you don't think this should ever happen, tell us why not.
- How does your AI player take into account the fact that the other player wants the AI player to lose? Does your strategy work even if two AI players are playing against each other?

Third, make sure you understand how the support code works and fits together.

- How do you plan to complete the implementation of HumanPlayer?
- How does the Referee code work? You'll be asked to walk the TA through a brief explanation of the Referee code, to prove that you understand it.

This is the minimum that we're expecting, but feel free to do more! Remember, this is your opportunity to have the TAs look closely at your code, and to ask them questions about it.

Before your design check meeting, upload your notes and your `GameName.re` file to Gradescope. You must upload your files before your design check, even if that slot is before the official Gradescope deadline. Only one partner should hand in the files.

During the design check, you will discuss your solutions to each of the questions above with a TA. Please come prepared! The better prepared you are, the more productive the design check will be. Please also come on time — arriving late to your design check or final grading may result in a deduction.

7.2 Final Handin

The final hand-in is due by 7:00 PM, Dec 5, 2019.

For the final hand-in, you are required to hand in eight files: a '`README.txt`' file, either a '`Connect4.re`' or '`Mancala.re`' file containing your choice of game implementation, a '`HumanPlayer.re`' file, containing your human player implementation, a '`AIPlayer.re`' file, containing your AI player implementation (including minimax), a '`Referee.re`' file, containing the Referee module, a '`SigGame.re`' file containing the game signature, a '`SigPlayer.re`' file containing the player signature, and a '`CS17SetupGame.re`' file.

In the '`README`' file, you should provide:

- instructions for use, describing how a user would interact with your program (how would someone play your game against a friend? against the AI?)
- an overview of how your program functions, including how all of the pieces fit together
- a description of any possible bugs or problems with your program
- a list of the people with whom you collaborated
- a description of any extra features you chose to implement

Hand in your files via Gradescope. Only one partner has to submit the project. As always, be sure to inform your grader if you intend to use your late pass on this assignment!

7.3 Grading

The design check counts for 20% of your grade. Specifically,

- Your draft implementation of a `Game` module: 10 points
- Your plans for implementing the `HumanPlayer` and `AIPlayer` module: 7 points
- Your walkthrough of the Referee module: 3 points.

Functionality counts for 60% of your score. Specifically, we will look for:

- An initial state with customizable parameters

- Correctly generating legal moves
- Correctly determining the status of the game
- Correctly transitioning to the next state
- Handling human input and completing a human player
- Estimating the value of a game state well

Your AI Player will count for 20% of your grade. Specifically, we will look for:

- A correct implementation of minimax
- Making intelligent moves

Partial functionality merits partial credit.

You can lose up to 20% of your grade for bad style. Remember to follow the design recipe; that is, include type signatures, specifications, and test cases for all procedures your write. And add comments to any code which would otherwise be unclear.

8 Connect 4 Tournament (Optional)

At the end of this project, the TAs will be running an automated tournament pitting your Connect 4 AIs against each other in a mind-bending computerized showdown! Participation in the tournament is completely voluntary, but if you'd like to do so, you must

- Choose the game Connect 4
- Represent your `move` type as an integer between 1 and the number of columns in the board, 1 indicating a move into the leftmost column, and each successive integer indicating a move into the next column to the right
- Begin your Connect4 by defining initialRows to be 5 and initialCols to be 7, and using those variables in your inistialState instead of hard-coding them.

If there are enough Mancala submissions, we will also have a Mancala tournament. In order to participate in the tournament, your Mancala module should start with

```
let initialHoles = 6;
```

and the initialState should refer to initialHoles, and not hard-code the constant.

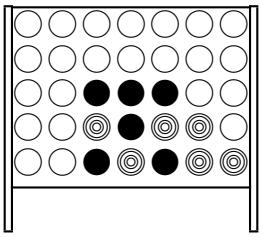
A The Games

A.1 Connect 4

Connect 4 is a game of brutal conflict. If you choose to implement this game, you'll be able to pit the skills you've honed for years against the cold heart of an AI player.

Connect 4 is played on a grid. One player is red and the other black. The players take turns dropping red and black checkers into the grid. The grid can be filled only from the bottom up. The red (black) player's goal is to line up four of red (black) checkers in a row.

Figure 1: A 5×7 Connect 4 game. Black has a guaranteed win.



Parameters While standard Connect 4 is played on a 6×7 board (but we refer to the board in this handout as 5×7 to make your AI move faster because they have to search less possible game states), you should make the size of your board configurable, allowing for play on any board with at least 4 rows and at least 4 columns. If you want an extra challenge, you can also make configurable the number of consecutive checkers required to win.

Strategy The Connect 4 strategy can be summarized fairly simply: always try to extend a line, and if possible, to leave both ends of the line open. If you can manage to make 3-in-a-row with empty slots at both ends, you have a guaranteed win. More generally, if with one move you can complete two different 3s-in-a-row, each with an empty slot at one end (and the empty slots are distinct), you're likewise guaranteed to win. As a special case of this, the trademark Connect 4 strategy is the one where you set up the two empty slots to be vertically adjacent (as in Figure 1); thus in blocking one win, your opponent sets you up for another!

A.2 Mancala

Mancala is one of the oldest known games that is still widely played today! There are a lot of variations, but here is the version we'd like you to implement:

The Mancala board is made up of two rows of 6 holes each. Four pieces/marbles are placed in each of the twelve holes. Each player is assigned to one row. Each player also has a store; these are placed on opposite ends of the board, and the store to the right of a player's row is their store.

The game begins with one player picking up all of the pieces in any one of the holes on their side of the board. Moving counter clockwise, the player deposits one piece in each hole until the stones run out.

If the player runs into their own store, deposit one piece into it. If they run into their opponent's store, skip it.

If the last piece you drop is in your own store, you get a free turn.

If the last piece you drop is in an empty hole on your side, you capture that piece and any pieces in the hole directly opposite. Captured pieces go into the store.

The game ends when all six holes in one row of the board are empty. The player who still has pieces on their side of the board when the game ends captures all those pieces.

The winner is the player with the most pieces in their store!

The video here shows some gameplay in action!

Parameters Most versions of Mancala are played with two rows of 6 holes, with 4 pieces/marbles starting in each hole. However, you should make both of these parameters configurable.

Strategy Look for opportunities to end a turn in your store, as this both gives you a point and allows you to take a free turn. For example, in your first turn, you'd want to start in the hole four spaces away from your store, as this will land your last piece in your store.

You also want to create empty holes on your side of the board, which will create more opportunities to capture your opponent's stones by ending a turn in one of them.

A.3 Other Games

Dissatisfied with the choices we've given you? Feel free to implement something else. The only restriction is: the game you pick has to be of the right type—that is, it has to be a two-player, sequential, finite-action, deterministic, zero-sum game of perfect information—and it must be nontrivial. Other acceptable examples include Chess, Gomoku, Pentago, Othello, and Dots and Boxes. What should guide your decision is your ability to code the game you choose and the corresponding AI in the time frame allotted.

B Other Optimization Methods

These extra optimizations are not required by any means; if you finish early and want to do more optimization, feel free to use these as a starting point.

Next State One optimization method for your AI player may rely on your nextState procedure.

Key idea: You want to be sure that you are looking at every possible game board when building your game tree.

When your AI player is making a move, it should think: if I move here, this will be the next state of the game board; now from this next state, here are all the moves that I can make, and if I move here then this will be the subsequent next state. By looking at every possible game board and every possible combination of moves, your estimateValue procedure will be more effective when assigning a move to a specific value as it will know the best combination of moves that can be played.

Alpha-Beta Pruning This method is all about short-circuiting using the minimax algorithm.

Key idea: You don't need to know values of all nodes to choose a move.

Alpha-Beta Pruning is an optimization method for the minimax algorithm. It greatly reduces the computation time, which would allow you to search much faster and much deeper in the game tree. This is important because you don't want your AI to take too long when making a move, but with alpha-beta pruning you can do much more in the same amount of time. It cuts off branches in the game tree that will not be selected because there already exists a better move available. To implement this, you will need to pass two additional arguments to minimax, alpha and beta.

Alpha Pruning for `minimum` This is a way of creating a short-circuiting *minimum* procedure. The trick here is that the procedure will take an additional argument, traditionally called α.

- **Input:** float α, procedure `f` list $[x_1, x_2, ..., x_k]$.
- **Output:** minimum among $[fx_1, fx_2; ...; fx_k]$ if minimum is greater than α, else $-\infty$.

The logic behind this is that if you already know that overall value is at least α, then any node with value less than α is irrelevant. This value can help you prune other nodes.

Beta Pruning for `maximum` This is a way of creating a short-circuiting *maximum* procedure. The trick here is that the procedure will take an additional argument, traditionally called β.

- **Input:** float β, procedure `f` list $[x_1, x_2, ..., x_k]$.
- **Output:** maximum among $[fx_1, fx_2; ..., fx_k]$ if minimum is greater than β, else ∞.

The logic behind this is that if you already know that overall value is at least β, then any node with value less than β is irrelevant. This value can help you prune other nodes.

Programming Strategy In order to implement alpha-beta pruning, you must keep track of both α and β and short-circuit both the max and min in the game tree.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: `http://cs.brown.edu/courses/csci0170/feedback`.