Lab 11: GoFirst and Nim
12:00 PM, Nov 18, 2019

Contents

1 Prologue 1

2 Game Theory 1

3 Game Signature 2

4 Nim, A Game Module 4

5 Let’s Play! 5

6 Just for Fun: AI 6

1 Prologue

While the big bad wolf was off taking a nap, the three little piggies decide to pass the time by playing some snazzy terminal games developed just for them. Unfortunately, they are pigs, and most pigs do not know how to program. The one that lives in a brick house asks you to help them by writing a game for them to play: Number in the Middle (Nim).

Before we begin PLEASE COPY OVER ALL OF THE FILES FROM /course/cs017/src/lab11.

2 Game Theory

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

- 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 their 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 solve a game using the minimax algorithm, together with a helper procedure that estimates the values of intermediate game states.

GoFirst and Nim are both examples of two-player, sequential, finite-action, deterministic, zero-sum games of perfect information.

GoFirst is essentially Tic-tac-toe on a one-by-one grid. On each turn, a player has the option to either go or pass. As you may have guessed, because there is only one space to fill, the first player to play go will win.

GoFirst is implemented for you in the lab slides. PLEASE REFER TO THE SLIDES AS A TEMPLATE FOR YOUR NIM GAME MODULE.

For the CS 17 version of Nim, the players take turns removing matches from a pile of 21 matches. On each turn, the player whose turn it is takes one, two, or three matches. The player who takes the last match loses.

In this lab, you will implement a simple version of Nim, with an AI player that doesn’t even need to use minimax to play intelligently.

## 3 Game Signature

First, you will design a game abstract data type (ADT). Your ADT will capture (much of) what is common to two-person, sequential, finite-action, deterministic, zero-sum games of perfect information.

Game Signature  Here is a start at a Game signature:

```ocaml
module type Game = {

  /* specifies a player */
  type whichPlayer = P1 | P2;

  /* status of game: if it's over (and who won) or ongoing (and who's turn ) */
  type status =
    | Win(whichPlayer)
    | Draw
    | Ongoing(whichPlayer);

  /* the state of the game: the position, status, anything else associated with the game at a given turn */
  type state;

  /* describes a move that a player can make */
  type move;
}
```
This signature has three main components:

- **Types**: What does a player look like? What about a state, or a move?
  - player: Names the two players.
  - state: Expresses the states of the game. What does the board look like, and whose turn is it?
  - move: Describes the game’s moves.

- **Constructor**: How do you construct a game? Specifically, what is the initial game state, before any moves have been made?
  - initialState: The initial game state.

- **Game logic**: These are the operations that define your game.
  - legalMoves: Produces all the legal moves at a given state.
  - nextState: Given a state and a move, produces the state that results from making that move.
  - estimateValue: Associated with each state in a zero-sum game is a value. However, it is not always possible to compute such a value. For example, the value of the initial state in a two-player, zero-sum game is either $+\infty$ (a win for P1) or $-\infty$ (a win for P2), but in Chess, this value is unknown. The procedure estimateValue computes an estimate of how good each state is. The lecture notes will more clearly explain this concept and how to approach writing this method.
Your first task in this lab is to extend this signature to include other useful features, such as a way of prettily printing a game.

For instance, in order to print out a specific player you could use `let stringOfPlayer : whichPlayer => string`.

**Task:** Add `stringOfPlayer`, `stringOfState`, `stringOfMove`, and `moveOfString` to your game signature with the correct types.

---

### 4 Nim, A Game Module

The brick-house piggie wants you to write a module that implements this game signature. Specifically, you should implement the game of Nim.

Here’s a road map for how you might proceed:

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

```haskell
module Nim = {
  type whichPlayer = P1 | P2;

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

  type state = ...
  type move = ...

  ...
};
```

For now, leave off the signature ascription, meaning `: Game`, so that your code will compile.

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

What are the states in a game of Nim? To answer this, think about what you would need to record if you were in the middle of a game of Nim, and you intended to take a break and finish the game later. Your `state` type should encapsulate just these things.

Once you have determined what is needed to represent a state in Nim, you can then go on to determine how to describe legal moves. That is, what will transition the game from one state to the next?

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 = failwith("not implemented");

4. The Nim module is concrete. You should leave it as such so that your helpers are fully visible, and you can test them.

But in addition, you should create a fully abstract Game module, right below the definition of Nim as follows:

```reason
module Game : Game = Nim;
```

This will ensure that your Nim module implements your game signature.

5. Gradually fill in all of the stubs you just wrote until they’re working.

At this point, two human players should be able to play your game. (See Section 6 for instructions.)

**Hint:** When writing estimateValue, think about in what cases you are guaranteed to either win or to lose. For instance, if you have 4 matches, is there any case when you can win?

| You’ve reached a checkpoint! Please sign up to get a lab TA to review your work.

5  Let’s Play!

Here is a player’s signature. All a player does is choose its next move.

```reason
module type Player = {
  /* given a state, and decides which legal move to make */
  let nextMove : Nim.state => Nim.move;
};
```

There are two kinds of players in this world: human players and AI players. Both can be found in the /course/cs017/src/lab11 directory.

1. The HumanPlayer makes its moves based on user input. We’ve written this player for you because it uses some imperative language constructs.

   To make this work, you need to be able to read in input from terminal and output to terminal. To do so, you need to install the readline-sync package. This is the part where you’ll actually change something other than a ReasonML file. Navigate to your project’s top-level directory, then install the ”readline-sync” package, which helps to supply a replacement for the Pervasives.read_line function that’s missing from the standard ReasonML installation.

   Then, using the “node package manager” program, npm, to install the aforementioned ”readline-sync” package by typing this into your Terminal:

   ```bash
   npm i --save readline-sync
   ```

   This puts a new piece of code somewhere that ReasonML will find it when building your project. And now ReasonML, and its surrounding “ecosystem”, will be able to build a program that includes something that replaces “read_line.”

---

1By the end of CS 18, you will be able to write a human player all by yourself!
2. The AIPlayer makes its moves based on artificial intelligence, which you will code soon.

The Referee module is in charge of running games. Play.re uses the Referee module to let you choose which kind of game to run and which players you’ll be playing with.

```re
/* The Referee coordinates the other modules. */
module R1 = Referee(Nim, HumanPlayer(Nim), AIPlayer(Nim));
```

It then invokes the playGame procedure (defined in the Referee module) to begin the game loop.

```re
R1.playGame()
```

**Task:** Play a few games of Nim against your partner. (Run node on Play.bs.js)

You’ve reached a checkpoint! Please sign up to get a lab TA to review your work.

### 6 Just for Fun: AI

Annoyed by the brick-house piggie winning all their games, the straw-house piggie wants a program that will help him always win. It is now up to you to write an AI that can defeat the brick-house piggie.

One nice thing about our simplified version of Nim is that there is a way to determine whether the current player will win or lose based solely on the number of matches left in the pile. What is this formula?

**Task:** Play Nim a few times and try to discover this formula. (Call over the TAs if you need help.)

Once you have the formula, you can define a perfect nextMove procedure: i.e., one that will always win if possible.

**Hint:** Don’t forget that you can use `estimateValue` to estimate how good a specific state within the game is.

**Task:** Write an AIPlayer module that implements the Player signature and always plays an optimal move.

**Hint:** Don’t worry! This is way easier than it seems. See the comments in AIPlayer.re to see exactly what you have to do.

Once a lab TA signs off on your work, you’ve finished the lab! Congratulations! Before you leave, make sure both partners have access to the code you’ve just written.

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](http://cs.brown.edu/courses/csci0170/feedback).