15-150 Summer 2017
Homework 09

Out: Saturday, 24 June 2017
Due: Monday, 26 June 2017 at 23:59 EDT

0 Introduction

In this homework you will implement a game of Tic-Tac-Toe and write an estimator for it, and create a Minimax player for the game.

0.1 Getting The Homework Assignment

The starter files for the homework assignment have been distributed through our `git` repository, as usual.

0.2 Submitting The Homework Assignment

Submissions will be handled through Autolab, at

`https://autolab.andrew.cmu.edu`

In preparation for submission, your `hw/10` directory should contain a file named exactly `hw10.pdf` containing your written solutions to the homework.

To submit your solutions, run `make` from the `hw/10` directory (that contains a `code` folder). This should produce a file `hw10.tar`, containing the files that should be handed in for this homework assignment. Open the Autolab web site, find the page for this assignment, and submit your `hw10.tar` file via the "Handin your work" link.

The Autolab handin script does some basic checks on your submission: making sure that the file names are correct; making sure that no files are missing; making sure that your code compiles cleanly. Note that the handin script is *not* a grading script—a timely submission that passes the handin script will be graded, but will not necessarily receive full credit. You can view the results of the handin script by clicking the number corresponding to the "check" section of your latest handin on the "Handin History" page. **If this number is 0.0, your submission failed the check script; if it is 1.0, it passed.**

Your `code/` folder must contain all the code that you want to have graded for this assignment, and must compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

0.3 Due Date

This assignment is due on Monday, 26 June 2017 at 23:59 EDT. Remember that you may use a maximum of one late day per assignment, and that you are allowed a total of three late days for the semester.

0.4 Methodology

You must use the five step methodology discussed in class for writing functions, for **every** function you write in this assignment. Recall the five step methodology:

1. In the first line of comments, write the name and type of the function.
2. In the second line of comments, specify via a `REQUIRES` clause any assumptions about the arguments passed to the function.
3. In the third line of comments, specify via an `ENSURES` clause what the function computes (what it returns).
4. Implement the function.
5. Provide test cases, generally in the format
   `val <return value> = <function> <argument value>`.

For example, for the factorial function presented in lecture:

```
(*  fact : int -> int
 *  REQUIRES:  n >= 0
 *  ENSURES: fact(n) ==> n!
*)

fun fact (0 : int) : int = 1
  | fact (n : int) : int = n * fact(n-1)

(* Tests: *)

val 1 = fact 0
val 720 = fact 6
```

0.5 Style

For this assignment, we will be grading your submissions based on your coding style. There are several ways that you can learn what is good style and what isn't:

- Your returned and graded homework submissions have been graded for style so use the markups for a reference.
- We have published solution code for the previous assignments, labs, and lectures.
- We have published a style guide at

  `https://www.cs.cmu.edu/~15150/resources/handouts/style.pdf`.

- You can ask your TAs about specific examples, or post on Piazza asking general questions.

If any code you submit for a problem violates our style guidelines or is otherwise difficult to understand, you can lose up to 4 points for that problem (that is, you can lose up to 4 points on Task 3.1, 4 points on Task 3.2, etc.). You can lose up to 10 style points overall on a homework assignment.

If you lose style points, you may present to a TA a compelling argument as to why your code has proper style, which, if successful, will grant you the points you lost due to style. Note that the TA you ask will have the final say in whether your code meets the style guidelines sufficiently.

1 Game 1: Tic-Tac-Toe

1.1 Introduction

You are on your way home having spent half of your day at Kennywood, and half of your day resolving agricultural conflict between Rohan and Vijay. Just as you make a turn onto your street, a wild Yue appears! He is the current world champion for TicTacToe, and has come specifically to challenge you, bearer of the powerful knowledge of Functional Programming. Can you create something with SML that demonstrates the power of programming languages and artificial intelligence, and beat him, the world champion, at Tic-Tac-Toe?

Your task is to create an estimator that can beat Yue. As attested to by his famous slogans "3×3 Tic-Tac-Toe is a Lie" and "Stay $n \times n$-ing," Yue will only play an estimator that allows for square boards of arbitrary size.

1.2 Tic-Tac-Toe

You are all likely familiar with this game, but for those who aren't:

Tic-Tac-Toe is a game where the goal is to mark any row, column, or either of the two diagonals of a given $n \times n$ board with all O's or X's. For this game, Maxie is O and Minnie is X, and Maxie always goes first. Any player may place their entry (and only their entry) onto an empty space on the board during their turn. Gameplay proceeds with each player taking a turn until the win conditions are met or until the board is filled. In any given game, your AI is either Minnie or Maxie, and Yue is the opposite.

Normally, when the board is filled, the game ends in a Draw. However, to simplify things, we'll say that Minnie wins when the board is filled to avoid having to clutter the signatures in your code with the extra Draw case. This ends up giving Minnie a huge advantage at the end of the day, but Maxie was being a bully anyways by wanting to go first all the time (what a jerk!).

2 Tic-Tac-Toe: Implementation of Types

2.1 State

For our implementation, an entry can either be an O or an X. The board is made up of tiles, with each tile being an entry option (`NONE` means there is an empty spot on the board). We represent the state of the game as the current board and whose turn it is.

```
datatype entry = O | X
type tile = entry option
type board = tile seq seq
type state = board * player
```

2.2 Moves

A location on the board can be easily represented as a pair of integers `(r,c)`, where `r` is the row and `c` is the column. Note that these are 0-indexed! A move is a tuple containing the location to place an entry on as well as the entry itself.

```
type location = int * int
type move = entry * location
```

WARNING: Do not change the type definitions/aliases mentioned on this page. Your code will fail to compile on Autolab if you do so and you will receive no credit for this part of the assignment.

2.3 Initial Board Size

The initial board size information can be retrieved from the options handed to the functor `TicTacToe`. The options structure ascribes to a signature `TTTCONSTS`, which contains a value `board_size`, a positive integer that determines the number of rows and columns. Use this information to help you generate the start state and test for valid moves.

3 Tic-Tac-Toe: Tasks

Task 3.1 (2 pts). In `game/tictactoe.sml`, create the `start` state with the info in `Settings`.

Task 3.2 (2 pts). Implement the function `is_valid_move : state * move -> bool`.
This takes in a state and move, and returns true if the move can be made on the board given in the state and false otherwise.

Task 3.3 (5 pts). Implement the function `make_move : state * move -> state`.
This, when given a state and move, will make the move, assuming the move is valid.

Task 3.4 (8 pts). Implement the function `moves : state -> move seq`.
This, when given a state, evaluates to a sequence of possible moves for the current player.

Task 3.5 (10 pts). Implement the function `status : state -> status`.
This, when given a state, will evaluate to `In_play` if the game can still be continued and `Over(outcome)` if the game is over with outcome `outcome`.

Task 3.6 (10 pts). Implement `estimate : state -> Est.est`.
`estimate` returns an estimated score for a given state.
Some quick rules about programming this function:

- `estimate` must **NOT** ever create nor examine future states. You may simulate moves at most one turn ahead as long as you never explicitly create a state (but you'll probably not need to do that for this game).
- `estimate` must **NOT** ever manipulate or hold stateful information. If you don't know what I'm referring to here, you're probably fine; this is for those people who might be thinking of using references, etc.

Failure to abide by these rules results in an automatic 0 for this task. If you are confused by what this means, please don't hesitate to ask on Piazza!

You will receive full credit as long as `estimate` performs no worse than an estimator that returns the maximum length of a chain of O's or X's for a particular player. Of course, a more refined evaluation algorithm will yield a more competent and challenging computer opponent! Your implementation of `estimate` should work on all valid game states, whether it is in play or it is a terminal state (leaf in the search tree). Remember that relatively lower scores should indicate Minnie is winning and relatively higher scores should indicate Maxie is winning. Your `estimate` should reflect this.

Please describe your implementation and the strategies you considered in a few sentences in a comment.

For testing, you can play the game in the REPL (see `lib/runtictactoe.sml`):

```
- CM.make "sources.cm";
- TicTacToe_HvH.go();
```

4 Views

Introduction to Views

As we have discussed many times, list 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 perspective using the sequence operations we have provided, you would have to write code that would lose style points, such as:

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

We can solve this problem using a *view*. This means we put an appropriate datatype in the signature, along with corresponding functions that convert sequences to and from this datatype. This allows us to pattern-match on an abstract type, while keeping the actual representation abstract. For this assignment, we have extended the `SEQUENCE` signature with the following components to enable viewing a sequence like 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 `Nil` and `Cons` 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 *)
```

Note that the second argument to `Cons` is another `'a seq`, *not* an `lview`. 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()`.

5 Minimax Algorithm

In lecture, we implemented the MiniMax game tree search algorithm: each player chooses the move that, assuming optimal play of both players, gives that player the best possible score (highest for Maxie, lowest for Minnie). This results in a relatively simple recursive algorithm, which searches the entire game tree up to a fixed depth.

5.1 Tasks

We have provided starter code in `minimax.sml`. We want to return not just the value of a node, but the move that achieves that value, so that at the top we can select the best move.

This gives rise to the following type:

```
type edge  = Game.move * Game.est
```

Here `Game.est` is a datatype that models players winning by providing numerical estimates to indicate which player appears to have an advantage.

The function `search` computes the best edge, meaning a tuple consisting of a move to a child along with the value of that child.

To evaluate each node, it is necessary to search all of the node's children and use information from those children to help return an appropriate value for the node itself.

You may find it helpful to implement the following helper functions, but they are optional:

```
max : (edge * edge) -> edge
min : (edge * edge) -> edge
vmax : (Game.est * Game.est) -> Game.est
vmax : (Game.est * Game.est) -> Game.est
```

max and min evaluate to the maximum and minimum edge, respectively; vmax and vmin evaluate to the maximum and minimum estimated value, respectively.

We also provide the following helper functions

```
valueOf : edge -> Game.est
moveOf : edge -> Game.move
```

which return the estimated value of an edge and the move from an edge.

Task 5.1 (10 pts). Define the function

```
fun search (d : int) (s : Game.state) : Game.est = ...
```

that finds the best edge from the node for game state `s`, assuming search depth `d` > 0.

Task 5.2 (4 pts). Define the function

```
    fun next_move (s : Game.state) : Game.move = ...
```

that returns the best move going out from `s`. Recall that `next_move` is a function specified in the `PLAYER` signature. In order to implement this function, you should search using the initial search depth as specified in `Settings`.

Testing: You do *not* need to provide tests for your individual functions. You will however want to test your code yourself, by running some games. In your REPL, after

```
- CM.make "sources.cm";
```

you can run any of the structures in `runtictactoe.sml` by entering `[structure name here].go();`. Feel free to also create your own testing structures using the library we have provided you.