COMP 212 Spring 2025
Final Non-Collaborative Challenge Problems

To hand in this assignment, you should upload `final.sml` and written file `final.pdf` to your handin folder.

1 Honor Code Verification

Task 1.1 Sign the following honor code verification by writing your name and today's date at the top of `final.sml`.

> I have read, understood, and followed the Academic Integrity Policy at https://dlicata.wescreates.wesleyan.edu/teaching/fp-s25/policy.html
>
> In particular, all work submitted for this assignment is entirely my own and was not derived from the work of others, whether a published or unpublished source, a web site, a generative AI tool, another student, other textbooks, materials from another course, from prior semesters of this course, or any other person or program. I have not spoken to anyone besides the professor about this assignment.
>
> I understand that any suspected violations of this policy will be referred to the Honor Board, with a recommended penalty of a grade of F in the course.

2 Game

In this problem, you will write components of a 2-player deterministic (no randomness like dice rolls) zero-sum (if one player wins then the other player loses) game. In class, we discussed the following signature for games:

```
signature GAME =
sig

    type board

    val show_board : board -> string

    type state = board * Player.player
```

```
    val start : state

    val hash : state -> string

    val check_status : state -> Player.status

    type move

    val parse_move : state * string -> move option
    val show_move : move -> string

    val possible_moves : state -> move Seq.seq

    val make_move : state * move -> state

end
```

The type `board` represents a game board. The `show_board` function displays a board as a user-readable string.

A `state` is a pair of a board and a player, describing whose turn it is. The `start` state represents the beginning of a game. A player is

```
datatype player = X | O
```

The `check_status` function returns the status of a game state:

```
datatype status = Playing | Won of player | Draw
```

where `Playing` means that the game is still ongoing, `Won` means that a player won, and `Draw` means it was a tie.

The type `move` represents a game move. The function `parse_move` determines whether a given user input string represents a valid move in the given game state, and returns `SOME(m)` if so and `NONE` if not. The function `show_move` converts a move to a string for printing.

The function `possible_moves` enumerates all possible moves from a game state; this can be used to print the possible moves for a human player to choose from and for an AI player to search for the best move.

The function `make_move` updates a game state by making a move (which can be assumed to be valid for that state).

One function that we didn't discuss in class: the `hash` function should convert a game state to as short a string as possible, to be used as the key of a dictionary.

For your reference, an implementation of the subtraction game from class is included in the handout code. The board of this game is a number representing a pile of "coins"; the initial state is typically 21, and player X starts. At each turn, a player removes 1 or 2 or 3 coins from the pile (up to the current contents of the pile). The goal of the game is to take the last coin, leaving the other player with 0 coins left.

There are three **independent** tasks in this problem, so you can try each of them even if you don't finish the previous one.

To compile this assignment, use `CM.make "sources.cm"`. The CM file for this assignment assumes that your files look like this

```
some folder/src
some folder/hw10-handout
some folder/final-handout
```

and uses the sequence code from `src` and your dictionary solution from homework 10.

2.1 Implementation of Tic Tac Toe

Task 2.1 (15 pts). Write a `structure TicTacToe :> GAME` implementing the game tic-tac-toe. For full credit, your implementation should work for an $N \times N$ square game board, with N pieces in a row to win, though for the rest of this description we assume $N = 3$. The rules should be as follows: The game board consists of a 3x3 grid. Each square of the grid is either empty or contains an X or an O. At each turn, a player can place their mark (X or O) on any empty square. The first player to get 3 in a row (horizontally, vertically, or on either diagonal) wins, which ends the game. If all spots are non-empty but no player has 3 in a row, then the game ends in a draw.

Note on testing: if you want to test this part before writing the next part, I suggest temporarily removing the `:> GAME` signature ascription so that SMLNJ treats the types in your implementation as public information. Then in SMLNJ you can test the functions interactively. Some examples for `CountDown`:

```
- CountDown.parse_move (CountDown.start, "1");
val it = SOME 1 : int option

- CountDown.parse_move (CountDown.start, "a");
val it = NONE : int option- CountDown.parse_move (CountDown.start, "1");

- CountDown.start;
val it = (21,X) : int * Player.player

- Seq.tolist (CountDown.possible_moves(CountDown.start));
val it = [1,2,3] : int list

- CountDown.make_move(CountDown.start, 1);
val it = (20,O) : int * Player.player
```

The rest of the project (the AI player and the controller) refers to a module `Game :> GAME`, which is defined to be `CountDown` initially; to play tic-tac-toe instead you should change the definition of `Game` to be `TicTacToe`.

2.2 Controller

Task 2.2 (15 pts). Write a function

```
fun controler (s : Game.state) : unit
```

that implements the top-level input-output loop for any `GAME`.

Here is a sample top-level loop for the CountDown game:

```
Controller.go();
21 pieces left
Player X enter your move.  The possibilities are
  1, 2, 3,
[you type] 1

20 pieces left
Player O is choosing a move

17 pieces left
Player X enter your move.  The possibilities are
  1, 2, 3,
[you type] 2

15 pieces left
Player O is choosing a move

12 pieces left
Player X enter your move.  The possibilities are
  1, 2, 3,
[you type] 3

9 pieces left
Player O is choosing a move

8 pieces left
Player X enter your move.  The possibilities are
  1, 2, 3,
[you type] 2

6 pieces left
Player O is choosing a move

4 pieces left
Player X enter your move.  The possibilities are
  1, 2, 3,
[you type] 3
```

```
1 pieces left
Player O is choosing a move

0 pieces left
O wins!
```

The only input-output functions you should need are:

- `TextIO.inputLine(TextIO.stdIn) : string option` reads a line of input that the user types into the console every time you call it
- `print : string -> unit` prints the given string. You will want to include the newline ASCII character `\n` in your strings to make line breaks.

See Lab 10 / the controller from homework 6 for examples.

For this task, you can either make both players play by requesting human input (if you are not planning to do the AI player task, this way you can play against yourself or a friend), or you can make one player human and the other an AI. The function `Train.best_next_state` contains a stub AI player that just always picks the first available move.

2.3 AI Player

Write an AI player that can play a 2-player deterministic zero-sum game. Here is a suggested algorithm.

Associate a "score" with each state of the game. The score of a Win/Draw state is `1.0` if X won and `~1.0` if O won, and `0.0` if it is a draw. For the Playing states, start by assigning every state score 0.0, and then update these scores as the AI player plays many games against itself. You can store the updated scores in a dictionary mapping game states to scores. In the `Train` module, the type

```
type memory = (string,real) Dict.dict
```

represents such a dictionary. You can change the type of the keys if you want to, but for efficiency, you may want to use as keys a short string representation of the states determined by the `Game.hash` function.

Task 2.3 (22 pts). Write a function

```
best_netx_state : memory * Game.state -> Game.state * real
```

that is given a memory and a game state and chooses the best next state available by making one of the possible moves of the game from that state. `best_netx_state` should choose the best state using the scores learned so far and stored in the given memory (or the 1.0/0.0/-1.0 scores described above if the game is over). "Best" means that if the game state indicates that it is X's turn, the function picks the state with the highest score; if the game state indicates that it is O's turn, the function picks the next state with the lowest score. The function should return the best next state along with its score.

One subtlety is that there will often be more than one state with the same maximal score. For the player to learn to play from enough states, it is helpful to choose one of the tied states randomly, rather than picking a fixed one. You can use the function `randRange(low:int, high:int) : int` to choose a random number between low and hi (inclusive, assuming low $\leq$ high).

Task 2.4 (23 pts). Write a function

```
fun train (n : int, mem : memory, s : Game.state) : memory
```

that trains the player by making it play `n` turns of the game against itself, choosing the best next state at each turn, and updating the scores in the memory after each turn. The function should return the memory containing the final scores learned from this training. When in the process of training the game ends, the training continues with a new game on `Game.start`.

After each turn's move is chosen, the score of one state is updated as follows: when the AI player moves from a state s_1 to a state s_2, the memory is updated so that

$$\text{new score of } s_1 = \text{old score of } s_1 + 0.3(\text{old score of } s_2 - \text{old score of } s_1)$$

That is, the score of s_1 is nudged closer to the score of the state s_2 that the player moved to. The 0.3 is a "learning rate" that influences how "much" the player learns from a given move. In this way, the AI player "learns" how good or bad s_1 is, based on the states that it moves to next after s_1.

The line

```
val mem = Train.train(1000000, Dict.empty, Game.start)
```

in the `Controller` trains the player on 1,000,000 plays. For my implementation/computer this takes less than a minute, but you can decrease the number of plays if this is too slow on your computer.

3 Proof and Analysis

In the first lecture, we added a grid of numbers by first adding each row, and then adding the results: `sum (map (sum, c))`. An alternative and shorter implemenetation is to flatten the grid into a sequence of numbers, and then add that sequence: `sum(flatten c)`. In this problem, you will prove that these two pattern compute the same result.

Specifically, recall the tree type from Homework 7, and your implementations of `map`, `reduce`, and `flatten` for these trees on that homework.

Assume a function `n : 'a * 'a -> 'a` with a base case `e : 'a` and tree of trees `t : ('a tree) tree`.

Task 3.1 (15 pts). Prove that

```
reduce(n,e, map( fn t' => reduce (n,e,t'), t))
=
reduce(n,e, flatten t)
```

State any assumptions about `n` and `e` that the proof requires.

Task 3.2 (8 pts). Discuss whether the two sides of the equation have the same work, span, and memory usage, or whether you expect one side or the other to be more efficient. Consider both big-O and exact costs.

Task 3.3 (7 pts). Repeat the above analysis of the work, span, and memory usage interpreting the above code as acting on sequences instead of trees. That is,
for `t : ('a Seq.seq) seq`, compare the work and span and memory usage of
`Seq.reduce(n,e, Seq.map( fn t' => Seq.reduce (n,e,t'), t))`
and `Seq.reduce(n,e, Seq.flatten t)`.

You can assume that `Seq.flatten(s)` is implemented by

```
Seq.reduce(Seq.append,Seq.empty(),s)
```