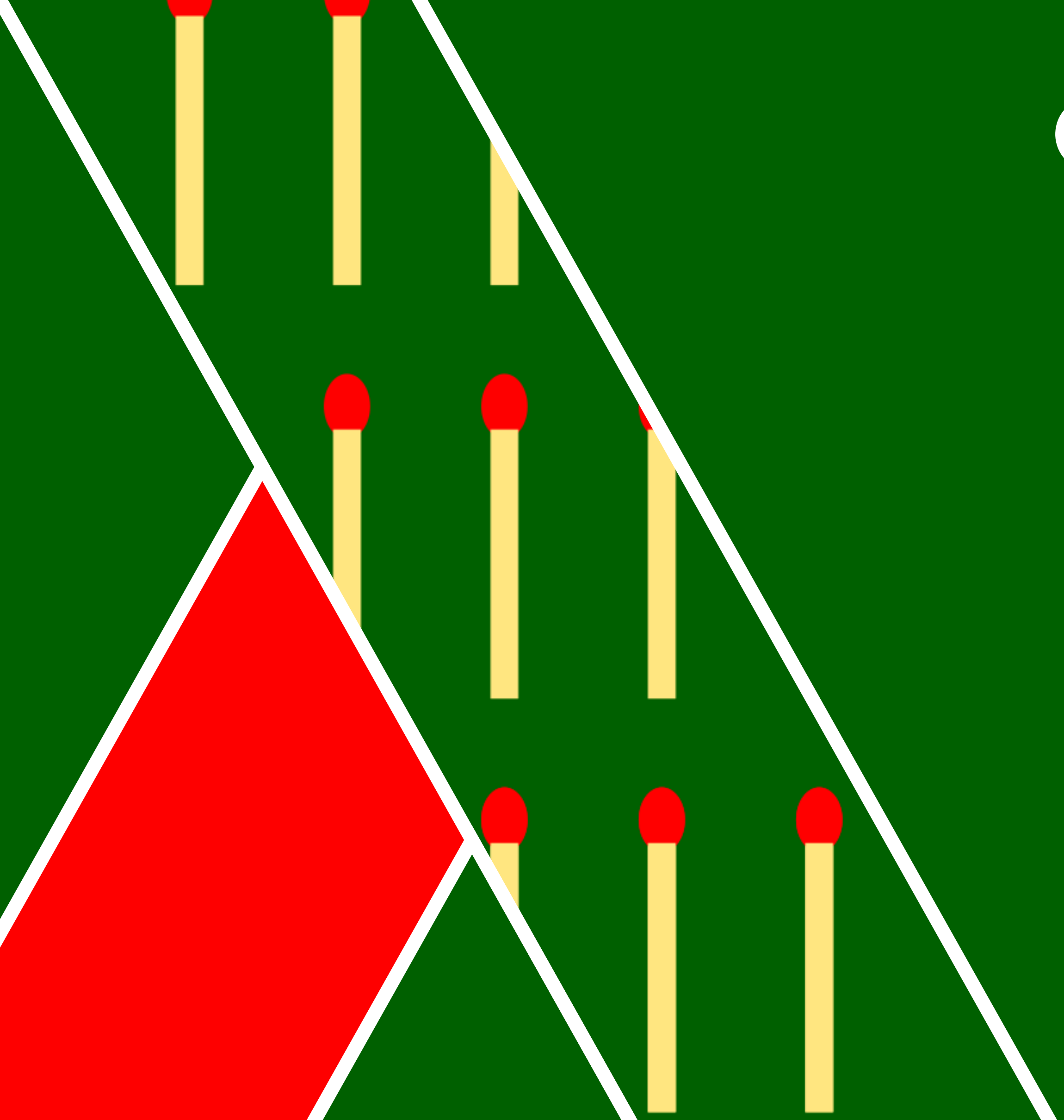

Games I: The Game Signature

15-150 M21

Lecture 0723
23 July 2021

Why is this in 150?

- **Modular Abstraction**: notice how the modules system allows us to logically structure our code, as well as enforce (and make use of) abstraction boundaries
- **Types**: Notice how we can use well-chosen `datatype`s and pattern-matching to write clear, human-readable, semantic code. This will be important when there's so many complex moving parts!

We've made a series of design decisions with how we structure our `GAME` code. There are many other ways to do it.

Let's play a game

The game *Nim* is played in the following way:

1. The game begins with a certain number of pebbles
2. The players take turns. On their turn, a player may take either 1, 2, or 3 pebbles.
3. The player who takes the last pebble **loses**

Demonstration: Nim Play

0 Implementing Games in SML

What kind of games?

The games we'll consider are:

- 2-player (with alternating turns)
- *Deterministic* (no dice)
- *Perfect information* (players do not have "private" information)
- *Zero-sum* (either one player wins and the other loses, or it's a tie)
- *Finitely-branching* (on each turn, a player only has finitely-many moves available to them)
    - Not the same thing as finite! There could be infinitely-many possible game states.

These are simplifying assumptions. We can drop some of them, but that would complicate our analysis.

The implementation of games we'll use revolves around 3 signatures

- `GAME` (Nim, Connect4, Tic-Tac-Toe, Checkers)
- `PLAYER` (Human player, directly-implemented player, MiniMax player, Alphabeta player)
- `CONTROLLER` (`Controller` functor)

Each `PLAYER` is playing a particular `GAME`. A `CONTROLLER` takes two `PLAYER`s playing the same `GAME`, and plays them against each other.

Documentation:

Game Reference

0723.0 (lib/game/core/SHOW.sig)

```
signature SHOW =
sig
  type t
  val toString : t -> string
end
```

Player

0723.1 (lib/game/core/Player.sml)

```
structure Player =
  struct

    datatype t = Minnie | Maxie
```

0723.2 (lib/game/core/Player.sml)

```
    val flip = fn
      Minnie => Maxie
    | Maxie  => Minnie
```

0723.3 (lib/game/core/GAME.sig)

```
signature GAME =
sig

  structure State   : SHOW  (* game states *)
  structure Move    : SHOW  (* moves       *)
  structure Outcome : SHOW  (* results *)
```

0723.3 (lib/game/core/GAME.sig)

```
    datatype status = Playing of State.t
                    | Done of Outcome.t

    exception InvalidMove of string

    val play    : State.t * Move.t -> status

    val player : State.t -> Player.t
    val moves  : State.t -> Move.t Seq.t

end
```

(code for `Nim : GAME`)

0723.4 (lib/game/core/PLAYER.sig)

```
signature PLAYER =
sig

  structure Game : GAME

  val next_move : Game.State.t -> Game.Move.t

end
```

Tuesday in lab, *you'll* write `NimPlayer :> PLAYER` which plays optimally.

0723.5 (lib/game/core/CONTROLLER.sig)

```
signature CONTROLLER =
sig

  structure Game : GAME

  val play : Game.State.t -> Game.Outcome.t

end
```

0723.6 (lib/game/core/Controller.fun)

```
functor Controller (
  structure Game    : GAME
  structure Player1 : PLAYER
  structure Player2 : PLAYER
  sharing Game = Player1.Game = Player2.Game
) : CONTROLLER =
struct
```

So now we can play...

How can we design interesting `PLAYER`s to play against us when the game is less tractable? How can we train them to play *intelligently*?

Next Week

Programming intelligent players

- Estimators
- The Minimax Algorithm
- Alpha-Beta Pruning

Thank you!