Modular Framework for the following kinds of games:

- **2-player**  (alternate turns)
- **deterministic**  (no dice)
- **perfect information**  (no hidden state)
- **zero-sum**  (I win, you lose; ties ok)
- **finitely-branching**  (maybe even finite)
Modular Framework for the following kinds of games:

- **2-player** (alternate turns)
- **deterministic** (no dice)
- **perfect information** (no hidden state)
- **zero-sum** (I win, you lose; ties ok)
- **finitely-branching** (maybe even finite)
- Examples: **tic-tac-toe, connect4, ...**
Example: Nim

• Take 1, 2, or 3 pieces of chocolate
• Alternate turns
• Player who leaves an empty table loses
Game Trees

- **Nodes** represent current state of game
- **Edges** represent possible moves
- A given **level** corresponds to a given player, alternating turns

  – Our players: **Maxie** and **Minnie**
Game Trees

- **Nodes** represent current state of game
- **Edges** represent possible moves
- A given **level** corresponds to a given player, alternating turns

– Our players: **Maxie** and **Minnie**

**Important:** These trees are not predefined datatypes, but instead are implicit representations of possible game evolutions. We will represent them functionally, expanding nodes as necessary using sequences to represent the result of possible moves.
A Nim Game Tree

Start with 4 pieces of chocolate
A Nim Game Tree

MAXIE moves first

4
A Nim Game Tree

MAXIE

3

take 1

4
A Nim Game Tree

MAXIE

Minnie moves

3

take 1

4
A Nim Game Tree

MAXIE

Minnie

take 1

take 1
A Nim Game Tree

MAXIE

Minnie

2  MAXIE  moves

3  take 1

4  take 1
A Nim Game Tree

MAXIE

Minnie

MAXIE

Minnie moves

1

take 1

2
take 1

3
take 1

4
A Nim Game Tree

MAXIE
Minnie
MAXIE
Minnie

0
1
2
3
4

take 1
take 1
take 1
take 1

A Nim Game Tree

MAXIE

Minnie

MAXIE

Minnie

MAXIE

Minnie

MAXIE moves
A Nim Game Tree

Maxie wins!
A Nim Game Tree

MAXIE

Minnie

MAXIE

Minnie wins!

MAXIE

Minnie wins!

MAXIE

Minnie wins!

MAXIE

Minnie wins!

MAXIE

Maxie wins!

MAXIE
Nim game tree with leaf values

- Purple circle means Maxie wins, assign value +1.
- Green circle means Minnie wins, assign value -1.
Now compute interior node values:

- A purple node means Maxie wins, assign value +1.
- A green node means Minnie wins, assign value -1.
Now compute interior node values:

- A purple circle means Maxie wins, assign value +1.
- A green circle means Minnie wins, assign value -1.
Now compute interior node values:

- Purple nodes mean Maxie wins, assign value +1.
- Green nodes mean Minnie wins, assign value -1.
Maxie can win!

The other two initial Maxie moves would allow Minnie to win.
Estimators

• In practice, trees are too large to visit leaves.
• Instead:
  – expand tree to some depth,
  – use game-specific estimator to assign values (not just ±1) at bottom-most nodes explored.
• Backchain mini-max values as before.
• Repeat after each actual move.
• Issue: horizon effect.
Estimators

• In practice, trees are too large to visit leaves.
• Instead:
  – expand tree to some depth,
  – use game-specific estimator to assign values (not just ±1) at bottom-most nodes explored.

Our simplified presentation associates the estimator with GAME. More generally, one would make it PLAYER-dependent.

Either way, our automated PLAYERS assume optimal play by both Maxie and Minnie relative to the estimator.
Nim has perfect estimator

Player making move can win for sure iff

\[ n \mod 4 \neq 1 \]

(n is number of pieces)

Why?
Nim has perfect estimator

Player making move can win for sure iff

\[ n \mod 4 \neq 1 \]

(n is number of pieces)

Why?

Player and opponent must each take 1, 2, or 3 pieces. Given player can ensure 4 pieces total are taken after player and opponent each has taken a turn. So, the player can always leave opponent with \(4k+1\) pieces (some \(k\)). Eventually opponent must take last piece.
Maxie can win!
Modular Framework

- **Game** : GAME (e.g., Nim : GAME)
- **Player** : PLAYER (includes a Game)
- **Referee** : GO (glues 2 Players to play)

- Will have automated and human players.
- Will write automated players as functors that expect a Game. Code plays without knowing Game details, except implicitly via estimator.
Modular Framework

Nim | Connect4 | Checkers | Chess | GAME | MiniMax | AlphaBeta | PLAYER | Human | Referee | VerboseRef

structures | signature | functors | signature | functors

(rough picture; there will be a few more administrative layers)
signature \texttt{GAME} = \texttt{sig}

\textbf{end}
signature GAME =
sig
datatype player = Minnie | Maxie

The concrete type player models a two-person game.

We call one player Minnie and the other Maxie, because we think of them as minimizing and maximizing values associated with nodes in a game tree (these values are based on some approximate estimator).
The concrete datatype `outcome` models the idea that either one of the players wins or there is a draw, once a game ends.
GAME Signature

signature GAME =
sig
    datatype player = Minnie | Maxie
    datatype outcome = Winner of player | Draw
    datatype status = Over of outcome | In_play

Finally, a game is either Over (with a given outcome) or still In_play.

The concrete datatype status models this aspect of the game.
The types \texttt{state} and \texttt{move} depend on the particular game being played, so we leave them abstract in the signature.
This line of the signature says that every particular game implementation must specify a value representing the start state of the game.
signature GAME =
sig
  datatype player = Minnie | Maxie
  datatype outcome = Winner of player | Draw
  datatype status = Over of outcome | In_play

type state (* abstract *)
type move (* abstract *)

val start : state

val moves : state -> move Seq.seq

(REQUIRE that the state be In_play
ENSURE that the move sequence is non-empty and all moves valid)
GAME Signature

signature GAME =

sig

datatype player = Minnie | Maxie

datatype outcome = Winner of player | Draw

datatype status = Over of outcome | In_play

type state (* abstract *)
type move (* abstract *)

val start : state

val moves : state -> move Seq.Seq

val make_move : state * move -> state

(REQUIRE that the move be valid at the state.)
signature GAME =
sig
  datatype player = Minnie | Maxie
  datatype outcome = Winner of player | Draw
  datatype status = Over of outcome | In_play

type state (* abstract *)
type move (* abstract *)

val start : state

val moves : state -> move Seq.seq
val make_move : state * move -> state

val status : state -> status
val player : state -> player

These functions are called “views”. They allow a user to see some information about the abstract type state. (Here, the player function returns the player whose turn it is to make a move.)
**GAME Signature**

signature GAME =

sig

datatype player = Minnie | Maxie
datatype outcome = Winner of player | Draw
datatype status = Over of outcome | In_play

type state (* abstract *)
type move (* abstract *)

val start : state
val moves : state -> move
val make_move : state -> move
val status : state -> status
val player : state -> player

datatype est = Definitely of outcome | Guess of int
val estimate : state -> est

(REQUIRE that the state be In_play)

(CAUTION: estimate need not provide useful info)

A more general approach would place the estimator in a separate module. It is here for presentational simplicity.
signature GAME =
  sig
    datatype player = Minnie | Maxie
    datatype outcome = Winner of player | Draw
    datatype status = Over of outcome | In_play

    type state (* abstract *)
    type move (* abstract *)

    val start : state

    val moves : state -> move Seq.seq
    val make_move : state * move -> state

    val status : state -> status
    val player : state -> player

    datatype est = Definitely of outcome | Guess of int
    val estimate : state -> est

    . . . (* functions to create string representations *)
end
Nim Structure

structure Nim : GAME =
struct
Nim Structure

structure Nim : GAME =
struct
    datatype player = Minnie | Maxie
    datatype outcome = Winner of player | Draw
    datatype status = Over of outcome | In_play
end

The types player, outcome, and status were specified in the GAME signature, so we need to write them, i.e., implement them, exactly as there.
We now implement the abstract type `state` as a particular datatype constructor expecting a pair. The pair specifies how many pieces are available and whose turn it is to take one or more pieces.

Recall: The player whose turn it is must take 1, 2, or 3 pieces, but not more pieces than are available. A player who takes all available pieces loses.

Why use constructor `State` rather than merely the pair `int * player`?

Ascription is transparent (one reason for that is to make it easier for us in this course to see what is happening when testing the code).

However, we do not want anyone messing with the internal representation even though they can see it. Since `State` is not specified in the signature, no one can pattern match on it.
Nim Structure

structure Nim : GAME =
struct
    datatype player = Minnie | Maxie
    datatype outcome = Winner of player | Draw
    datatype status = Over of outcome | In_play
    datatype state = State of int * player
    datatype move = Move of int
end

We implement the abstract type move as a datatype that specifies how many pieces to take.
Nim Structure

```ocaml
structure Nim : GAME =
struct
    datatype player = Minnie | Maxie
    datatype outcome = Winner of player | Draw
    datatype status = Over of outcome | In_play
    datatype state = State of int * player
    datatype move = Move of int

    val start = State (15, Maxie)
end
```

We can make this be any positive integer. We could even make it be an argument to a functor that creates a Nim structure. For simplicity, we make it 15 here.
Nim Structure

structure Nim : GAME =
struct
    datatype player = Minnie | Maxie
    datatype outcome = Winner of player | Draw
    datatype status = Over of outcome | In_play

    datatype state = State of int * player
    datatype move = Move of int

val start = State (15, Maxie)

fun moves (State (n, _)) =
    Seq.tabulate (fn k => Move (k+1)) (Int.min (n,3))

Create all valid moves at a given state (as a move Seq.seq) corresponding to taking 1 piece, 2 pieces, or 3 pieces, but no more than are still available. (We may assume there is at least 1 piece available.)
Nim Structure

structure Nim : GAME =
struct
    datatype player = Minnie | Maxie
    datatype outcome = Winner of player | Draw
    datatype status = Over of outcome | In_play

    datatype state = State of int * player
    datatype move = Move of int

    val start = State (15, Maxie)

    fun moves (State (n, _)) =
        Seq.tabulate (fn k => Move (k+1)) (Int.min (n,3))

    fun flip Maxie = Minnie
    | flip Minnie = Maxie

    fun make_move (State (n, p), Move k)= State (n-k, flip p)

We may assume the move is valid, so can simply subtract the number of pieces taken. And we change whose turn it is.
Nim Structure

structure Nim : GAME =
struct
  datatype player = Minnie | Maxie
  datatype outcome = Winner of player | Draw
  datatype status = Over of outcome | In_play
  datatype state = State of int * player
  datatype move = Move of int

  val start = State (15, Maxie)

  fun moves (State (n, _)) =
    Seq.tabulate (fn k => Move (k+1)) (Int.min (n,3))

  fun flip Maxie = Minnie
      | flip Minnie = Maxie

  fun make_move (State (n, p), Move k) = State (n-k, flip p)

datatype est = Definitely of outcome | Guess of int

(Type est was specified in the signature, so we need to write it as there.)

end
structure Nim : GAME =
struct
  datatype player = Minnie | Maxie
  datatype outcome = Winner of player | Draw
  datatype status = Over of outcome | In_play
  datatype state = State of int * player
  datatype move = Move of int
val start = State (15, Maxie)
fun moves (State (n, _)) =
  Seq.tabulate (fn k => Move (k+1)) (Int.min (n,3))
fun flip Maxie = Minnie
  | flip Minnie = Maxie
fun make_move (State (n, p), Move k) = State (n-k, flip p)
datatype est = Definitely of outcome | Guess of int
fun estimate (State (n, p)) =
  if n mod 4 = 1 then Definitely (Winner (flip p))
  else Definitely (Winner p)
end

Recall that Nim has a perfect estimator (generally a game will not).
structure VeryDumbNim : GAME =
struct
  datatype player = Minnie | Maxie
  datatype outcome = Winner of player | Draw
  datatype status = Over of outcome | In_play
  datatype state = State of int * player
  datatype move = Move of int
val start = State (15, Maxie)
fun moves (State (n, _)) =
  Seq.tabulate (fn k => Move (k+1)) (Int.min (n,3))
fun flip Maxie = Minnie
  | flip Minnie = Maxie
fun make_move (State (n, p), Move k) = State (n-k, flip p)
datatype est = Definitely of outcome | Guess of int
fun estimate _ = Guess 0

Of course, there is no requirement that the estimator be useful. We could trivialize it!
Nim Structure

structure Nim : GAME =
struct
  datatype player = Minnie | Maxie
  datatype outcome = Winner of player | Draw
  datatype status = Over of outcome | In_play
  datatype state = State of int * player
  datatype move = Move of int
val start = State (15, Maxie)

fun moves (State (n, _)) =
  Seq.tabulate (fn k => Move (k+1)) (Int.min (n,3))

fun flip Maxie = Minnie
  | flip Minnie = Maxie

fun make_move (State (n, p), Move k) = State (n-k, flip p)

datatype est = Definitely of outcome | Guess of int

fun estimate (State (n, p)) =
  if n mod 4 = 1 then Definitely (Winner (flip p))
  else Definitely (Winner p)
end
structure Nim : GAME =
struct
    datatype player = Minnie | Maxie
    datatype outcome = Winner of player | Draw
    datatype status = Over of outcome | In_play

    datatype state = State of int * player
    datatype move = Move of int

    ...

We have not yet implemented the two views, so let us do that now:
Nim Structure (cont)

structure Nim : GAME =
struct
    datatype player = Minnie | Maxie
    datatype outcome = Winner of player | Draw
    datatype status = Over of outcome | In_play

    datatype state = State of int * player
    datatype move = Move of int

    ...

    fun player (State (_, p)) = p

The player view of a state returns the player whose turn it is.
Nim Structure (cont)

structure Nim : GAME =

struct
  datatype player = Minnie | Maxie
  datatype outcome = Winner of player | Draw
  datatype status = Over of outcome | In_play

  datatype state = State of int * player
  datatype move = Move of int

  ...

  fun player (State (_, p)) = p

  fun status (State (0, p)) = Over (Winner p)
  | status _ = In_play

The status view of a state checks whether there are any pieces remaining. If so, the game is In_play. If not, then the previous player must have taken all the remaining pieces, Therefore, the current player is the winner.
Nim Structure (cont)

```haskell
structure Nim : GAME =
struct
    datatype player = Minnie | Maxie
    datatype outcome = Winner of player | Draw
    datatype status = Over of outcome | In_play

    datatype state = State of int * player
    datatype move = Move of int

    ...

    fun player (State (_, p)) = p

    fun status (State (0, p)) = Over (Winner p)
    | status _ = In_play

    ...

    (* functions to create string representations *)
end
```
signature PLAYER =

sig

  structure Game : GAME (* parameter *)
  val next_move : Game.state -> Game.move

end
We simply wrap one layer around the **GAME** signature, now requiring a function that decides what move to make given a particular game state.

In the next lecture we will write some automated game playing code. Here we show a simple interface that allows a human to play.
functor HumanPlayer (G : GAME) : PLAYER =

The code we write must provide a structure satisfying the PLAYER signature (think of that as an interface for playing games) that will work with any game G satisfying the GAME signature.
functor HumanPlayer (G : GAME) : PLAYER =
struct
  structure Game = G
end

The PLAYER signature requires a Game structure and a next_move function.
The game is the argument G passed to the functor.
functor HumanPlayer (G : GAME) : PLAYER =
struct
structure Game = G

(* read : unit -> string option *)
(* parse : G.state * string option -> G.move option *)

Next we need to write some functions to help with I/O. (These are not visible outside the structure created.)

Here we simply give the types of these functions.
functor HumanPlayer (G : GAME) : PLAYER =
struct
  structure Game = G

(* read : unit -> string option *)
(* parse : G.state * string option -> G.move option *)

reads from TexIO

Next we need to write some functions to help with I/O. (These are not visible outside the structure created.)

Here we simply give the types of these functions.
functor HumanPlayer (G : GAME) : PLAYER =
struct
    structure Game = G

    (* read : unit -> string option *)
    (* parse : G.state * string option -> G.move option *)

    fun next_move

end

Now we can write next_move.
functor HumanPlayer (G : GAME) : PLAYER =
struct
    structure Game = G

    (* read : unit -> string option *)
    (* parse : G.state * string option -> G.move option *)

    fun next_move s =
        let
            val _ = ... (* ask human to enter move *)
        in
            case parse(s, read()) of
                SOME m => m
                | NONE => next_move s (* for instance *)
        end
end
That is all.

See you Tuesday.

We will discuss the MiniMax and $\alpha \beta$ algorithms for automated game playing.