today

• playing **games** to win
  • modular programming
  • functors and code re-use
  • programming with sequences
games

- two players, taking turns  
- no randomness  
- players see everything  
- if I win, you lose  
- finitely many moves  
- No infinite move sequences  

2-person  
deterministic  
perfect information  
zero-sum  
finitely branching  
terminating
Simple Nim

- Start with a pile of sticks
- In each turn, take 1, 2 or 3 sticks
- Whoever takes the last stick loses
the plan

A framework for *game playing*

- signatures, structures, functors
- GAME, PLAYER, EVENT, REFEREE

Main example: **Simple Nim**

*Other design choices are possible*
*You will explore some in lab and homework*
Games

• A game has states and moves
• Making a move takes you to a new state
• Two players alternate
• Terminal states have no moves
• Terminal states have a score or payoff
A Nim game tree

Starting with 3 sticks, Me first

Nodes are states

Edges are moves

Leaf nodes are terminal states
strategy

picking moves that lead to the best outcome

If I take 1, you can take 1, then I have to take 1 and lose.
Starting from 3 sticks, I have a best move.

I’ll take 2, then You must take 1 and lose.
strategies

• A strategy is a function from states to moves
  • A winning strategy for Me means I can win, no matter what you do
• Games don’t always have winning strategies...
assessing outcome

For each state, we can compute an outcome (or label) that predicts the best possible result achievable from that state, assuming that both players try their best.

- Since Nim is a zero-sum game, we use +1 for “I win” and -1 for “You win”.
  - I want to maximize the outcome.
  - You want to minimize the outcome.
Nim tree analysis

We can compute labels for states, using bottom-up propagation

• Leaf nodes are easy (the last player loses)

• At a Me node, if any child has label $+1$ label this node as $+1$; otherwise use $-1$

• At a You node, if any child has label $-1$ label this node as $-1$; otherwise use $+1$
game tree analysis

We can compute labels for states, using *bottom-up propagation*

- Leaf nodes are easy (the last player loses)
- At a *Me* node, use *maximum* child label
- At a *You* node, use *minimum* child label

*This works in general, not just for Nim!"
Nim analysis

- Label the leaf nodes

labels

+1: I win

-1: You win
Nim analysis

• Propagate

labels

+1: I win
-1: You win

Me (max)
You (min)
Me (max)
You (min)
Nim analysis

• Propagate again

labels
+1: I win
-1: You win
Nim analysis

- Propagate again

labels

+1: I win
-1: You win
Nim analysis

- ... propagate all the way to the root

labels
+1: I win
-1: You win

Me (max)
You (min)
Nim conclusion

- Label of state 3 is +1
- I should pick the move take 2

labels
+1: I win
-1: You win

I can win from 3 by moving to a state labelled +1
minimax

- This algorithm is known as \textit{minimax}
- Makes sense for arbitrary games
- But other games aren’t so well behaved!
  - may have \textit{tied} states
  - game tree may be \textit{large}
  - game tree may be \textit{infinite}
more generally

• May need to make a guess

• Could make predictions such as
  “Definitely I win”, “Definitely You win”,
  “Definitely a draw”, or “Guess n” (n : int)

  • For n > 0, Guess n means “looks like I win”
  • For n < 0, Guess n means “looks like You win”

(we won’t explore these ideas in class)
our plan

• Signatures for games, players, arenas, referees
• A structure for Nim
• A functor that builds minimax players for a given game
• Later: bounded search and heuristics

We won’t actually build game trees
Instead we’ll use recursion…
sequences

signature SEQ =

sig
  type 'a seq
  exception Range
  val nth : int -> 'a seq -> 'a
  val length : 'a seq -> int
  val null : 'a seq -> bool
  val tabulate : (int -> 'a) -> int -> 'a seq
  val empty : unit -> 'a seq
  val map : ('a -> 'b) -> ('a seq -> 'b seq)
  val reduce : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a
  val reduce1 : ('a * 'a -> 'a) -> 'a seq -> 'a
  val mapreduce : ('a -> 'b) -> 'b -> ('b * 'b -> 'b) -> 'a seq -> 'b
end
reduce l

REQUIRES  n > 0, g associative
ENSURES  reduce l g \langle x_1,...,x_n \rangle = x_1 \ g \ ... \ g \ x_n

null

null \langle x_1,...,x_n \rangle = true \ if \ n=0
null \langle x_1,...,x_n \rangle = false \ if \ n>0
signature GAME =
sig

  exception Fail of string

  type state

  type move

  val score : state -> int

  val moves : state -> move Seq.seq

  val step : state * move -> state

end

Games have states, moves,…
structure Nim : GAME =

struct
  exception Fail of string
  type state = int      (* number of sticks left *)
  type move = int       (* take this many sticks *)

  fun score 0 = 1      (* player who moved last lost *)
  |    score _ = raise Fail "Game not over"
  end

  fun moves s =        (* can take up to 3 sticks *)
    let
      val n = Int.min(s, 3)
    in
      Seq.tabulate (fn x => n-x) n
    end

  fun step (s, m) = (* the state reached by making move m in state s *)
    if (s >= m) then (s - m) else raise Fail "illegal move"
  end

moves s = \{3,2,1\} for s>2
moves 2 = \{2,1\}
moves 1 = \{1\}
moves 0 = \{}
PLAYERS

• A player for a game is a function for choosing moves in states of that game

• An arena is a game with two players
signature PLAYER =
  sig
    structure G : GAME
    val player : G.state -> G.move
  end

signature ARENA =
  sig
    structure G : GAME
    val player1 : G.state -> G.move
    val player2 : G.state -> G.move
  end
Given a structure \textbf{Game : GAME}

For simplicity, assume we’ve \textbf{opened} \textbf{Game}

\begin{verbatim}
  type state
  type move
  exception Fail of string
  val moves : state -> move Seq.seq
  val score : state -> int
\end{verbatim}

We will write functions

\begin{verbatim}
  F : state -> int
  G : state -> int
\end{verbatim}

\[ F(s) = \text{best possible outcome for Me, Maxie} \]
\[ G(s) = \text{best possible outcome for You, Minnie} \]
mutual recursion

- $F$ and $G$ are **mutually recursive**

  $F$ calls $G$
  $G$ calls $F$

ML syntax

```ml
fun F s = ...G...
and G s = ...F...
```

To figure out my best score
I need to calculate yours,
and vice versa
the minimax idea

\( F \, s = \text{score} \, s \quad \text{if moves} \, s = \langle \emptyset \rangle \)

\( F \, s = \text{maximum} \, \text{of} \, \langle G \, s_1, ..., G \, s_k \rangle \)
\[ \quad \text{if moves} \, s = \langle m_1, ..., m_k \rangle, \ k > 0 \]
\[ \quad \text{and} \quad s_i = \text{step}(s, m_i) \quad \text{for} \ i = 1 \ldots k \]

\( G \, s = \sim(\text{score} \, s) \quad \text{if moves} \, s = \langle \emptyset \rangle \)

\( G \, s = \text{minimum} \, \text{of} \, \langle F \, s_1, ..., F \, s_k \rangle \)
\[ \quad \text{if moves} \, s = \langle m_1, ..., m_k \rangle, \ k > 0 \]
\[ \quad \text{and} \quad s_i = \text{step}(s, m_i) \quad \text{for} \ i = 1 \ldots k \]
minimax functions

\( F, G : \text{state} \rightarrow \text{int} \)

fun \( F \) \( s \) =

\[
\text{let}
\quad \text{val} \ M = \text{moves} \ s
\quad \text{in}
\quad \text{if} \ (\text{null} \ M) \ \text{then} \ (\text{score} \ s) \ \text{else}
\quad \quad \text{reduce1 Int.max (map (fn m => G(step(s, m))) M)}
\quad \text{end}
\]

and

fun \( G \) \( s \) =

\[
\text{let}
\quad \text{val} \ M = \text{moves} \ s
\quad \text{in}
\quad \text{if} \ (\text{null} \ M) \ \text{then} \ \sim(\text{score} \ s) \ \text{else}
\quad \quad \text{reduce1 Int.min (map (fn m => F(step(s, m))) M)}
\quad \text{end}
\]
Nim

F 3 = best outcome for MaxiMe from state 3

moves 3 = 〈3, 2, 1〉

map (fn m => G(step(3,m))) 〈3, 2, 1〉 = 〈G 0, G 1, G 2〉

F 3 = reduce1 Int.max 〈G 0, G 1, G 2〉 = 1

G 0 = ~ (score 0) = ~1

G 1 = reduce1 Int.min 〈F 0〉 = 1

F 0 = score 0 = 1

G 2 = reduce1 Int.min 〈F 0, F 1〉 = ~1

F 0 = score 0 = 1

F 1 = reduce1 Int.max 〈G 0〉 = ~1

G 0 = ~ (score 0) = ~1
Nim

\[ F_3 = +1 \]
\[ G_0 = -1 \quad G_1 = +1 \quad G_2 = -1 \]
\[ F_0 = +1 \quad F_0 = +1 \quad F_1 = -1 \]
\[ G_0 = -1 \]

Game tree labelling

Recursion
best moves

• It’s easy to pick a best move from a state
  • a player should move to a state from which the outcome is optimal for them
• Easy to implement best move selection
  • Pair each potential move with the outcome it leads to
  • reduce (with min or max) to find the optimal move-outcome pair
  • then throw away the outcome
picking moves

a player (for MaxiMe) that seeks to maximize the outcome

definitions

type edge = move * int

fun max_edge ((m1,v1),(m2,v2)) = if v1 < v2 then (m2,v2) else (m1,v1)

fun maxbest (M : edge Seq.seq) = reduce1 max_edge M

fun player s = let
  val M = moves s
  val (m,_) = maxbest (map (fn m => (m, G(step(s, m)))) M)
in
  m
end

(only needed when moves s is non-empty!)
modularity

- This construction works for any structure Game : GAME
- We can encapsulate... as a functor

functor MiniMax(Game : GAME) : EVENT = ...

structure NimEvent = MiniMax(Nim)
functor MaxiMe(Game : GAME) =

struct
    structure Game = Game

    fun F s = let
        val M = Game.moves s
        in
            if (null M) then Game.score s else
                reduce1 Int.max (map (fn m => G(Game.step(s, m))) M)
        end
    and
        G s = let
            val M = Game.moves s
            in
                if (null M) then ~ (Game.score s) else
                    reduce1 Int.min (map (fn m => F(Game.step(s, m))) M)
            end
    end

    type edge = Game.move * int

    fun max_edge ((m1,v1),(m2,v2)) = if v1 < v2 then (m2,v2) else (m1,v1)

    fun maxbest (M : edge seq) = reduce1 max_edge M

    fun player s = let
        val M = Game.moves s
        val (m,_) = maxbest (map (fn m => (m, G(Game.step(s, m)))) M)
        in
            m
        end

end
- structure MaxieNimPlayer = MaxiMe(Nim);
structure MaxieNimPlayer : 
  sig
    structure Game : <sig>
      val F : Game.state -> int
      val G : Game.state -> int
      type edge = Nim.move * int
      val max_edge : ('a * int) * ('a * int) -> 'a * int
      val maxbest : edge seq -> Game.move * int
      val player : Game.state -> Game.move
  end

- MaxieNimPlayer.F 15;
  val it = 1 : int

- MaxieNimPlayer.player 15;
  val it = 2 : Nim.move

- MaxieNimPlayer.F 25;
  val it = ~1 : int

- MaxieNimPlayer.player 200;
  takes a LONG time

  Starting from 15, my best move is take 2 and I will win eventually

  Starting from 25, I will lose if my opponent is smart
more generally

• In many games the search tree is too large (maybe even infinite depth!)

• Can try minimax up to a fixed depth and make an estimate for deeper states

• Estimation may be based on a heuristic that predicts an outcome based on the current state
Nim heuristic

For Nim there is a **genius** heuristic

- In state $k > 0$ (with $k$ sticks remaining), the player to go next
  will **lose** if $k \mod 4 = 1$,
  will **win** otherwise

(we can prove this is 100% accurate,
assuming the player always chooses moves using this heuristic)
Nim theorem

- Let \( F, G \) be the Nim game functions.
- For all \( k > 0 \),
  
  \[
  F_k = \sim 1 \quad \text{if } k \mod 4 = 1
  
  F_k = +1 \quad \text{otherwise}
  \]

Proof?