today

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

- two players, taking turns
- no randomness
- players see everything
- if I win, you lose
- finitely many next 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 up to 1, 2 or 3 sticks
- Whoever takes the last stick loses
the plan

• A framework for *game playing*
  • signatures, structures, functors
  • GAME, PLAYER, …

• Main example: **Simple Nim**
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

Maxie

Minnie

Maxie

Minnie

Maxie

Minnie

Minnie
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

Maxie

Minnie

Maxie

Minnie
strategies

• A strategy is a function from states to moves
  • A winning strategy for Maxie means Maxie can win, no matter what Minnie does

• Games don’t always have winning strategies...
For each state in the game tree, we can compute a value (outcome) that predicts the eventual result from that state, assuming that both players try their best.

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

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

- Leaf nodes are easy (the last player loses)
- At a **Maxie** node, if any child has label +1, label this node as +1; otherwise use -1
- At a **Minnie** 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 *Maxie* node, use *maximum* child label
- At a *Minnie* 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

Me (Maxie)
You (Minnie)
Me (Maxie)
You (Minnie)
Nim analysis

• Propagate

labels

+1: I win
-1: You win

Me (Maxie)
You (Minnie)

Me (Maxie)
You (Minnie)
Nim analysis

- Propagate again

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

- Propagate again

labels

+1: I win
-1: You win

Me (Maxie)  
You (Minnie)

Me (Maxie)  
You (Minnie)
Nim analysis

- ... propagate all the way to the root

labels
+1: I win
-1: You win

Me (Maxie)
You (Minnie)
Me (Maxie)
You (Minnie)
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 minimax
- Makes sense for arbitrary games
- But other games aren’t so well behaved!
  - may have tied states
  - game tree may be large
  - game tree may be infinite
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)
Modular Framework

- Game : GAME (e.g. Nim)
- Player : PLAYER (includes a game)
- Referee : GO (glues 2 players to play)
our plan

• Signatures GAME, PLAYER,

• Structure NIM

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

  ...

end

continued on the next page
signature GAME =
sig
  ...
  val status : state -> status
  val player : state -> player

datatype est = Definitely of outcome | Guess of int

  val estimate : state -> est

  val stateToString : state -> string
  val statusToString : status -> string
  val estToString : est -> string
  val movesToString : move Seq.seq -> string
end
structure Nim : GAME =
struct
  datatype player = Maxie | Minnie
  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 flip Maxie = Minnie
     | flip Minnie = Maxie

  ...
end

continued on the next page
struct

fun make_move (State (n, p), Move k) =
    if (n >= k) then State (n - k, flip p)
    else Fail "tried to make an illegal move"

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

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

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

...
struct

…

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)

(* If there are n pebbles left, with n=1 (mod 4),
then the player whose turn it is must lose.
Otherwise, that player can win.*)

…

end
-Nim.estimate (Nim.start)

val it = Definitely (Winner Maxie)
signature PLAYER =
sig

structure Game : GAME

(* REQUIRES: Game.status(s) = ln_play *)
(* ENSURES: next_move(s) evaluates to a Game move legal at s. *)
val next_move : Game.state -> Game.move

end
signature SETTINGS =
sig
    structure Game : GAME
    val depth : int
end

To be used in bounding the depth of the game tree
Modular Framework

- Game : GAME (e.g. Nim)
- Player : PLAYER (includes a game)
- Referee : GO (glues 2 players to play)
signature TWO_PLAYERS =
sig

structure Maxie : PLAYER
structure Minnie : PLAYER
sharing type Maxie.Game.state = Minnie.Game.state
sharing type Maxie.Game.move = Minnie.Game.move
end

Any structure matching TWO_PLAYERS must be constructed so that Minnie and Maxie play the same game
Next time

- Define a functor that creates a MiniMax player
  - expects a structure ascribing to SETTINGS and produces a structure ascribing to the signature PLAYER

- Define a functor Referee that takes two players and produces a structure to run the game
  - expects a structure ascribing to TWO_PLAYERS and produces a structure that can be used to run the game
functor HumanPlayer (G : GAME) : PLAYER =
struct
structure Game = G

fun readmove () =
case TextIO.inputLine TextIO.stdIn of
    NONE => raise Fail "early input termination; aborting"
  | SOME(str) => SOME(String.substring(str, 0, String.size(str)-1))
(* This strips off a trailing newline character, as required by G.parse_move. *)

fun parsemove(state, NONE) = NONE
  | parsemove(state, SOME(str)) = G.parse_move state str

fun player_to_string (G.Maxie)  = "Maxie"
  | player_to_string (G.Minnie) = "Minnie"

fun next_move state =
  let
    val _ = print(player_to_string(G.player state) ^ ", please type your move: ")
  in
    case parsemove(state, readmove()) of
      SOME(m) => m
    | NONE => (print "Something is wrong; bad input or bad move.\n";
               next_move state)
  end
end (* HumanPlayer *)
fun parse_move (State (n, _)) str =
  let
    fun enough k = if k <= n then SOME(Move k) else NONE
  in
    case str of
      "1" => enough 1
    | "2" => enough 2
    | "3" => enough 3
    | _   => NONE
  end