Introduction to Games

15-150

Principles of Functional Programming

Slides for Lecture 23

December 2, 2025

Adapted from slides by

Michael Erdmann

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

4

Start with 4 pieces of chocolate

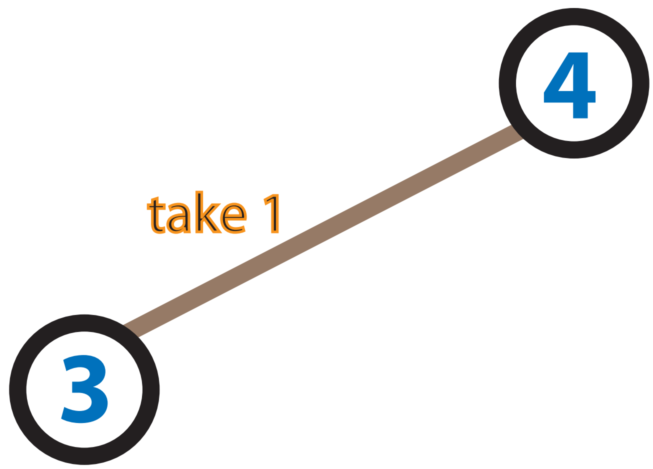
A Nim Game Tree

MAXIE moves first

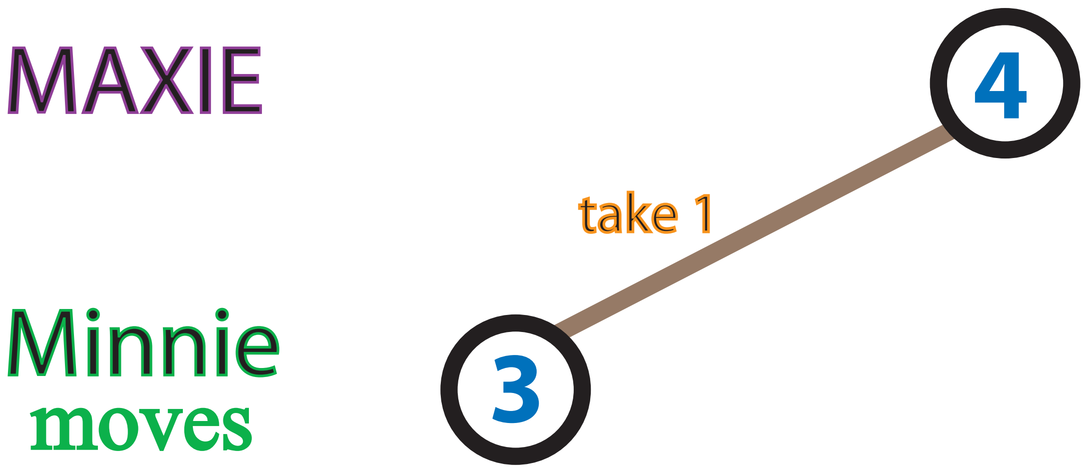
4

A Nim Game Tree

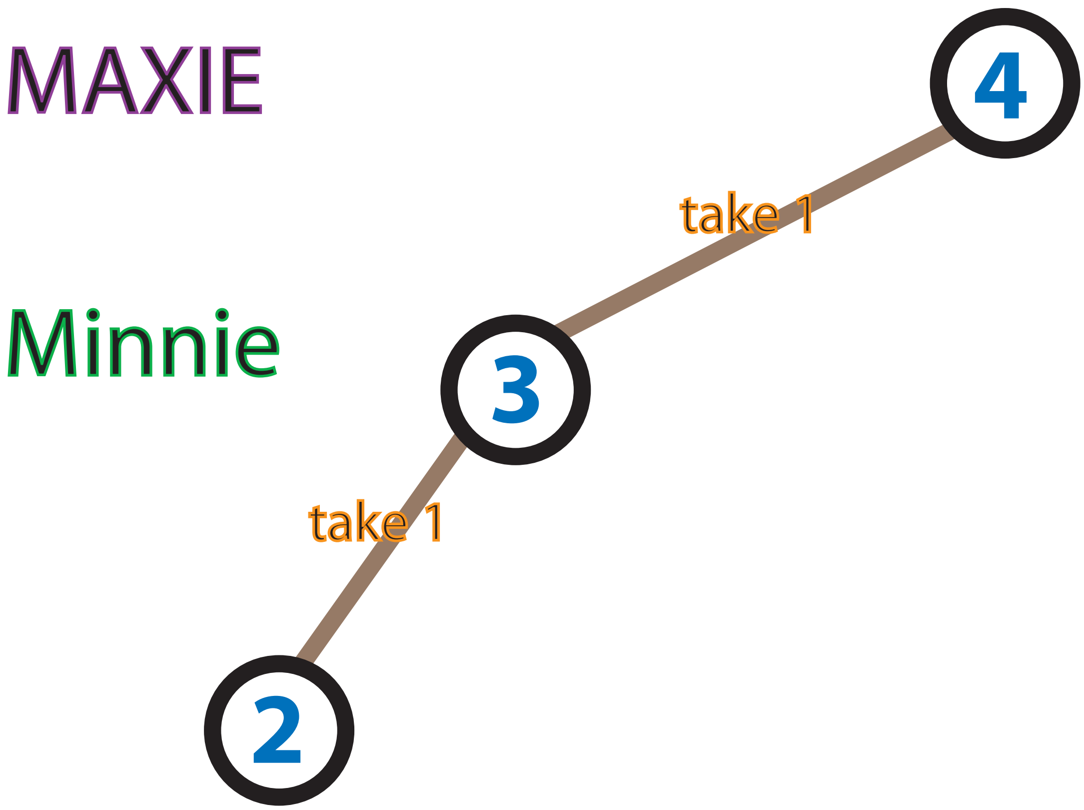
MAXIE



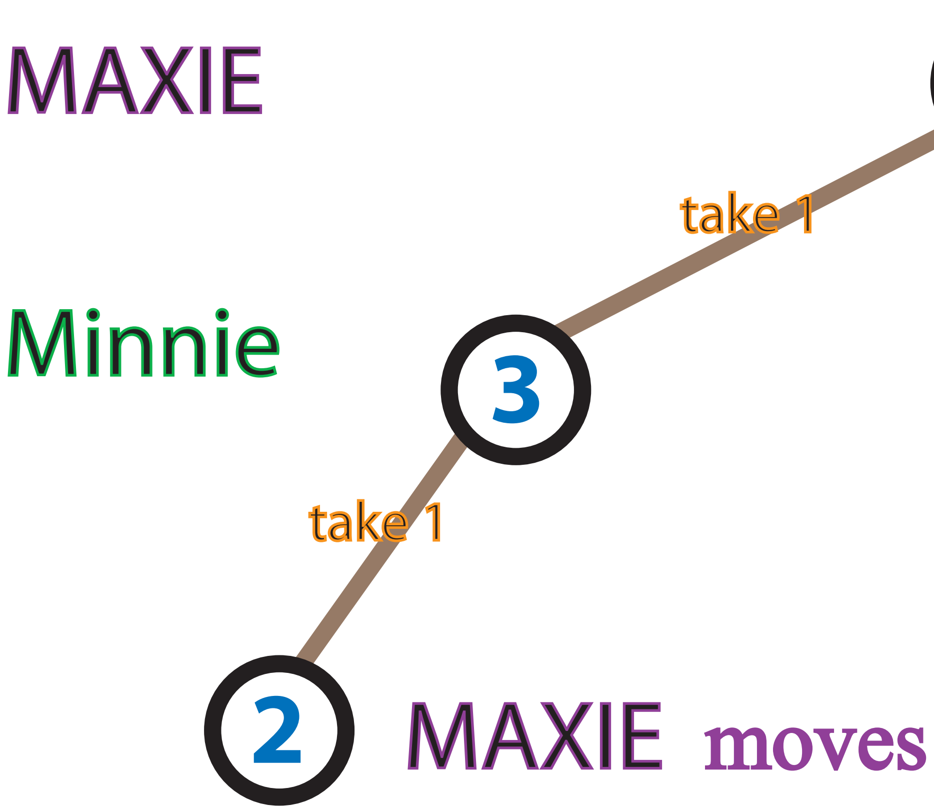
A Nim Game Tree



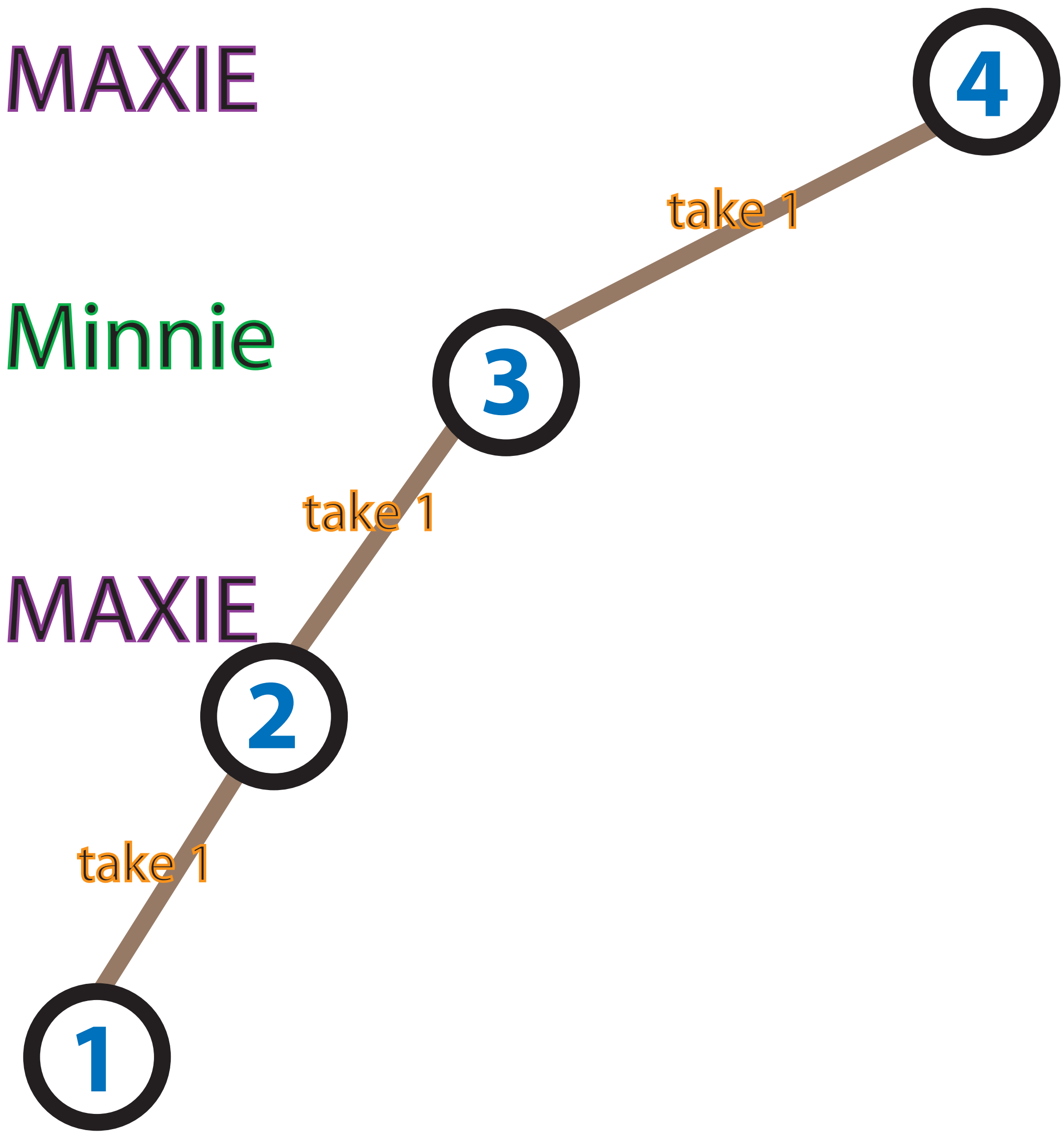
A Nim Game Tree



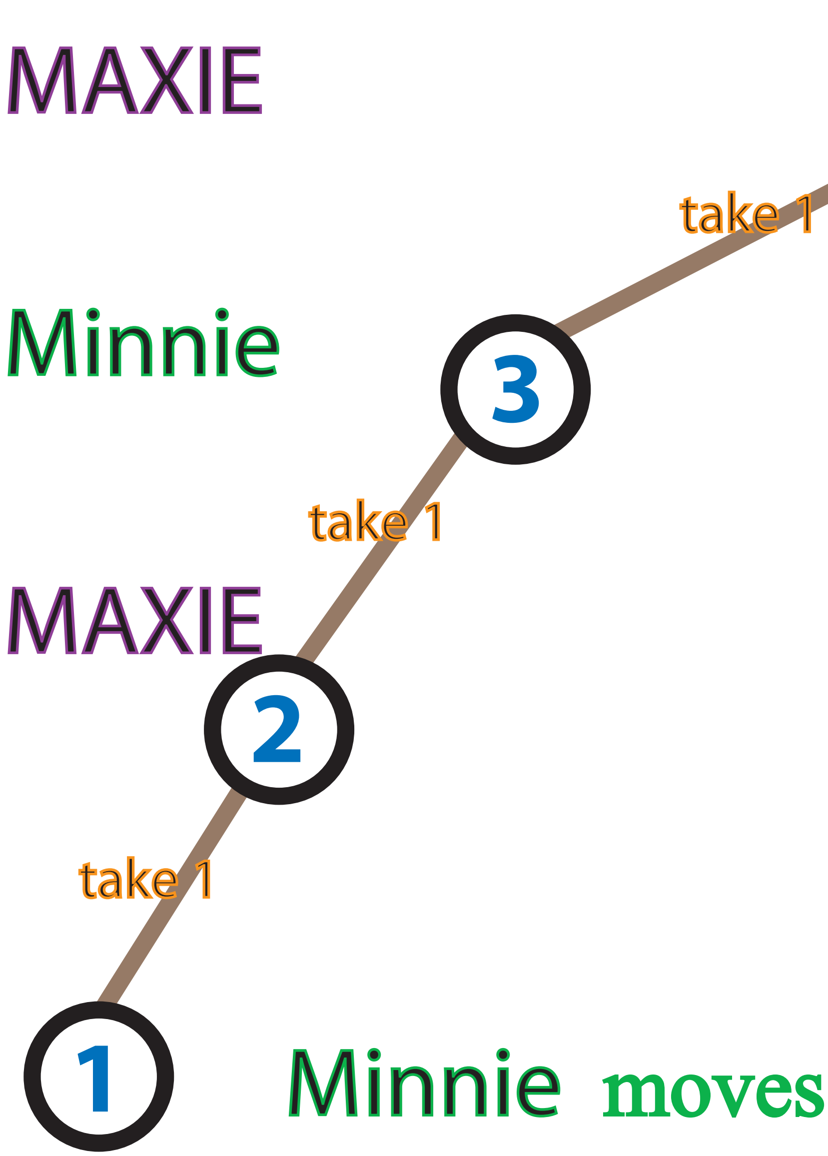
A Nim Game Tree



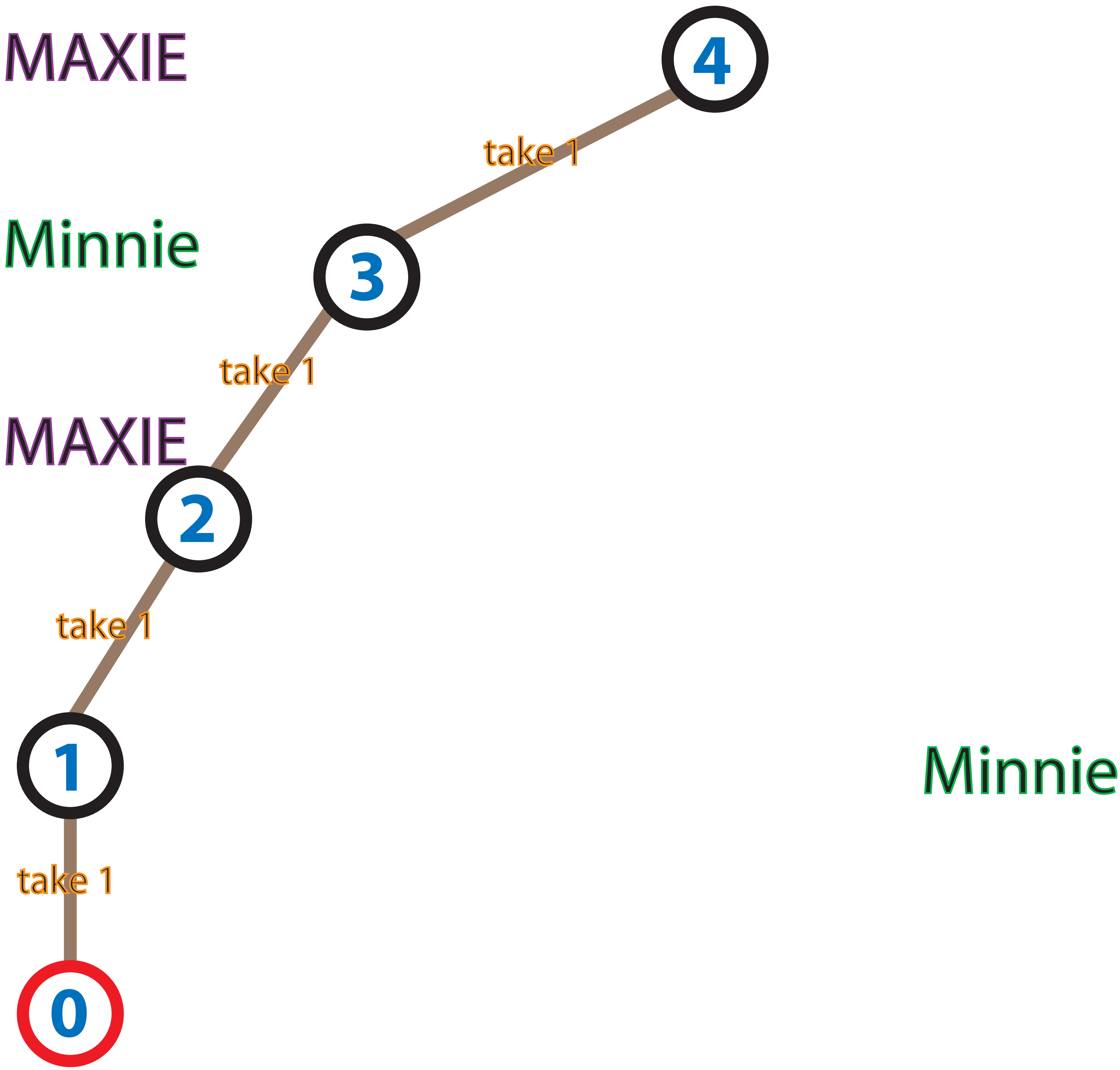
A Nim Game Tree



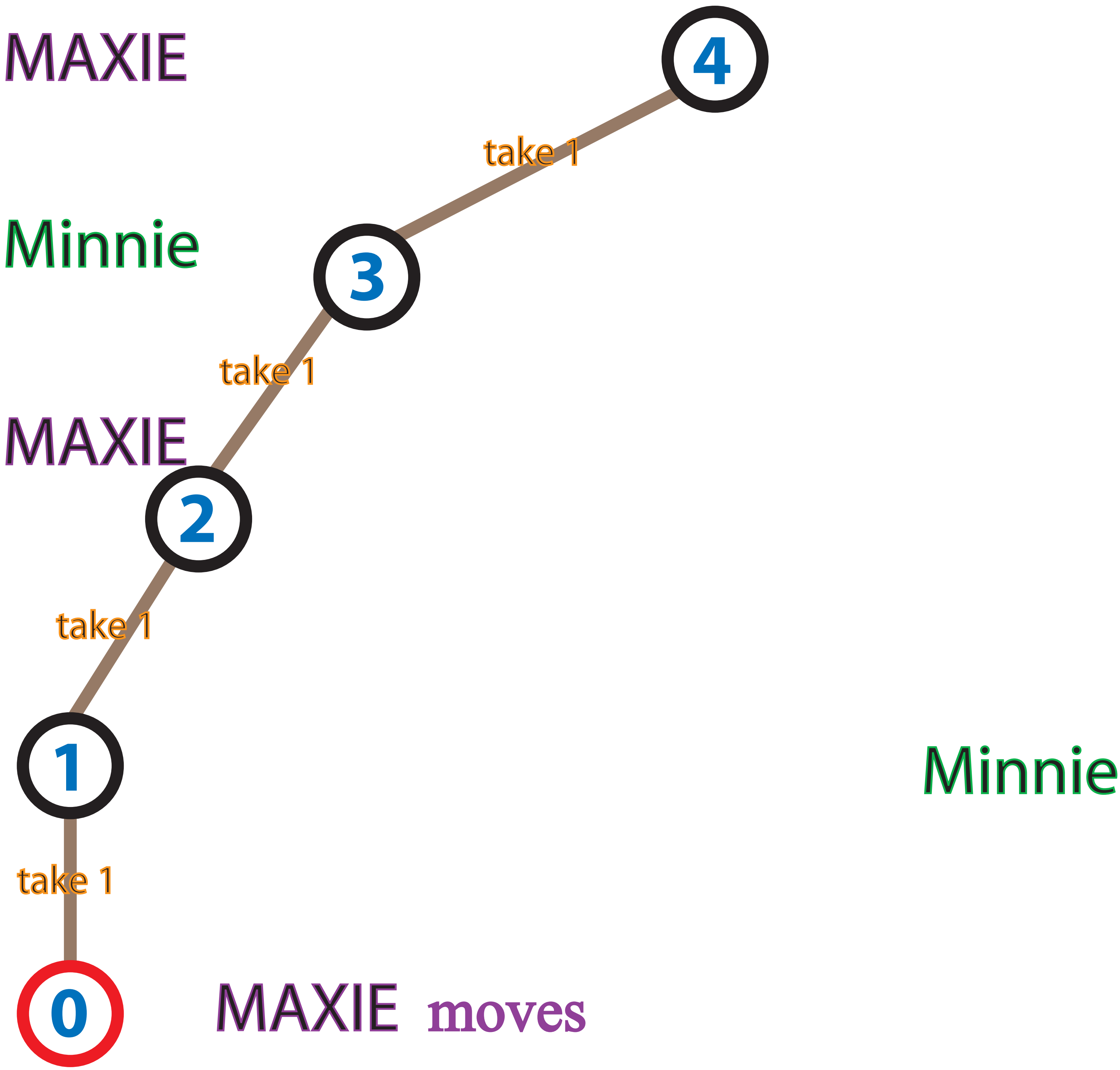
A Nim Game Tree



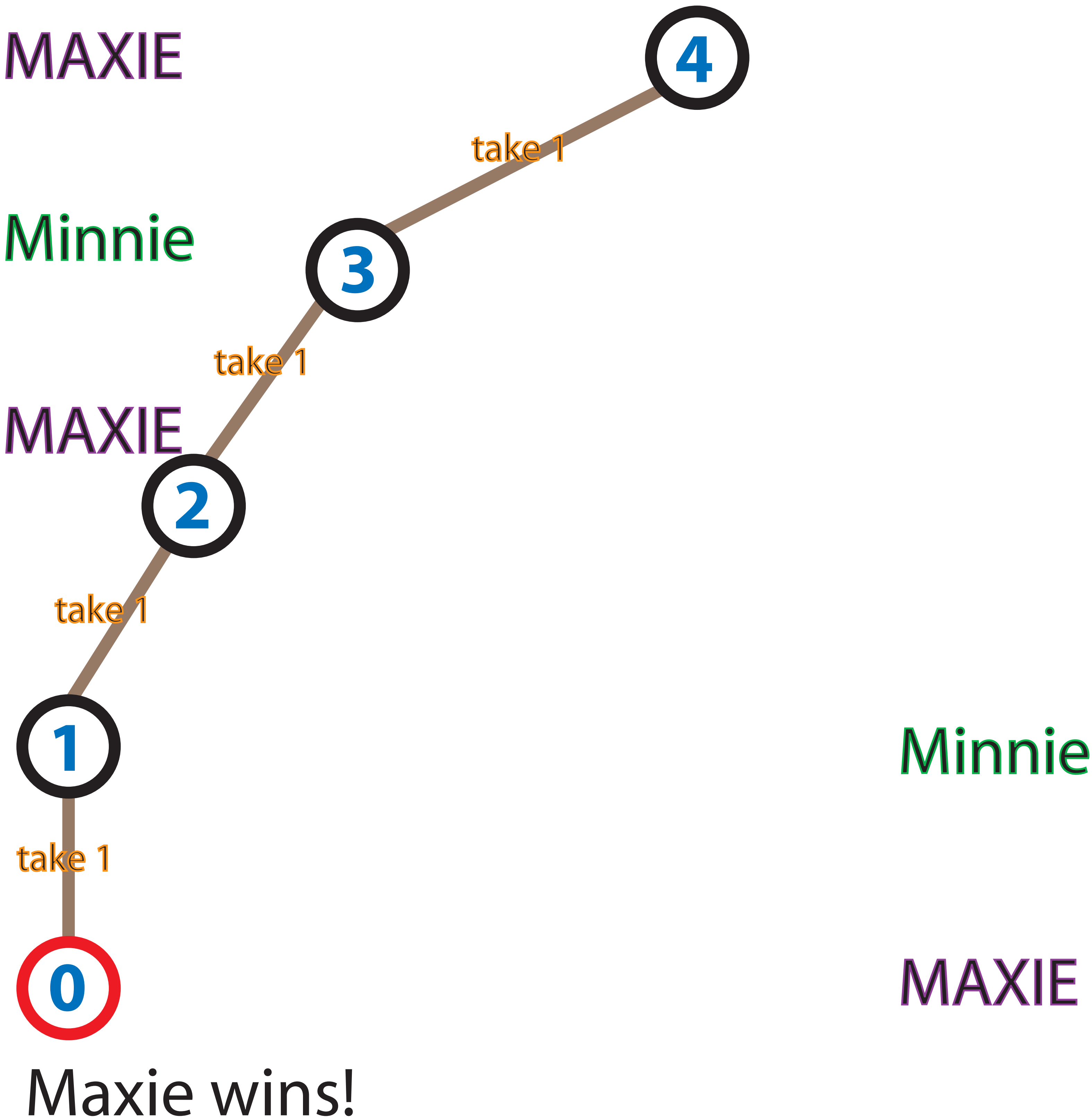
A Nim Game Tree



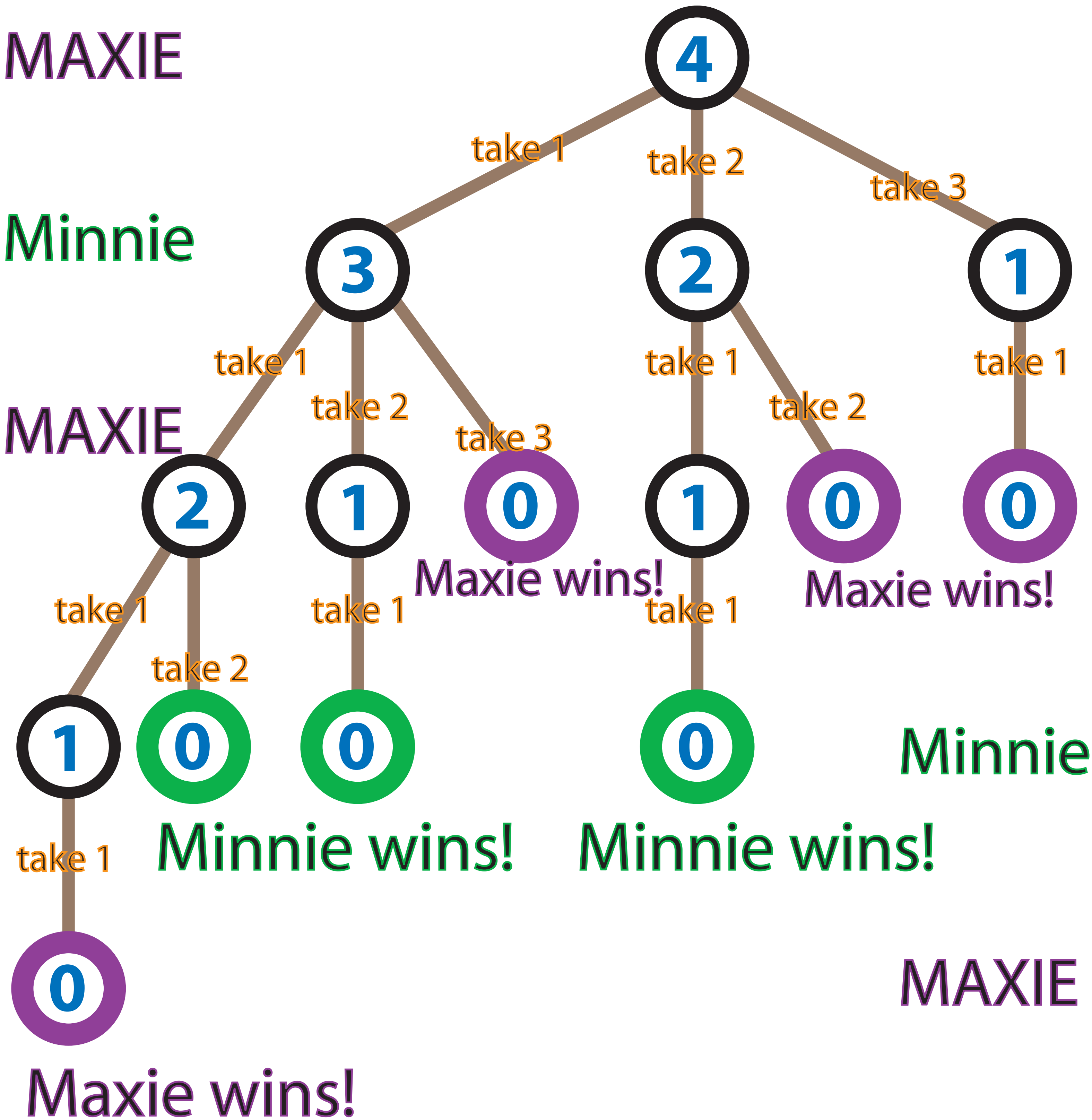
A Nim Game Tree



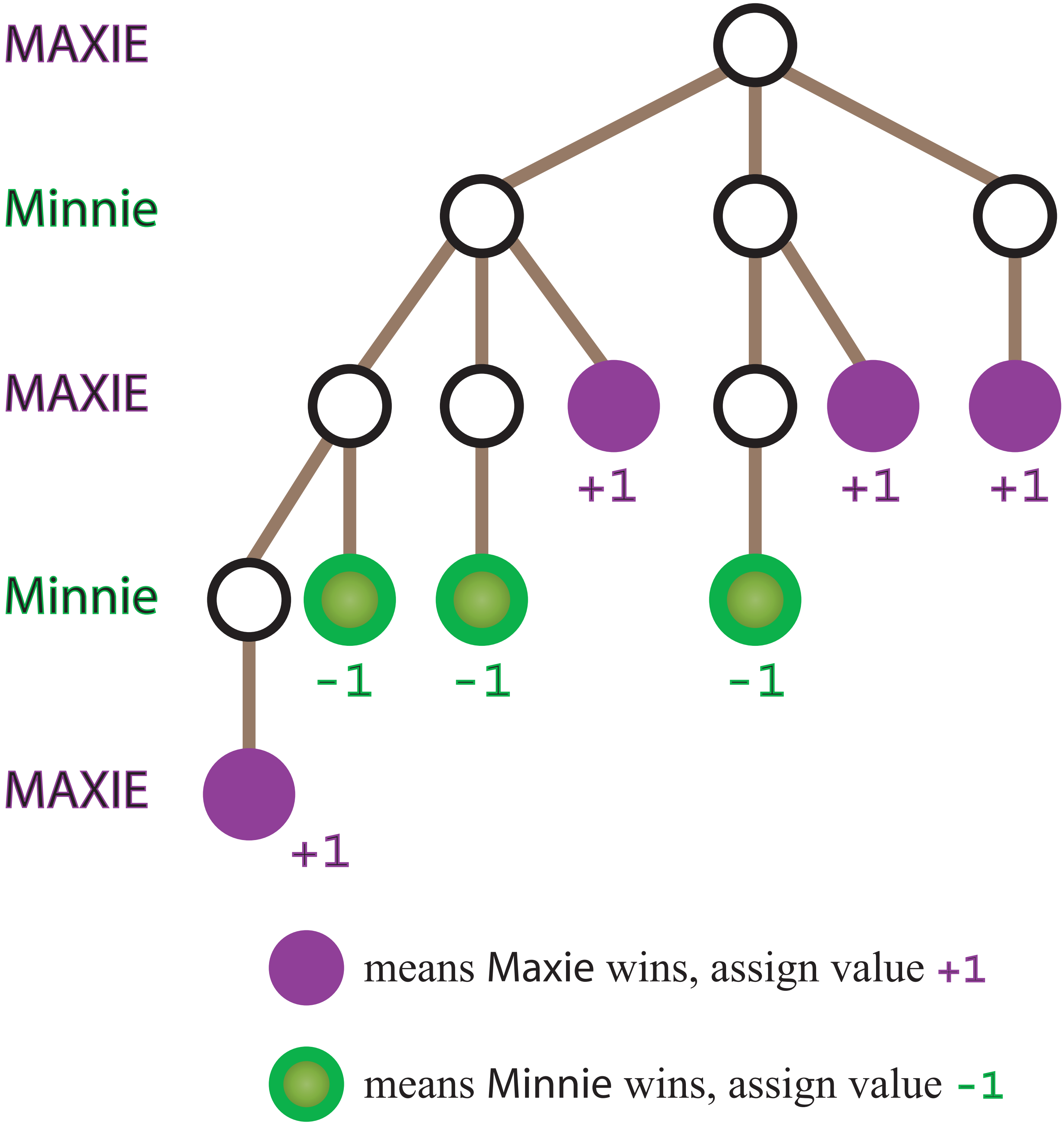
A Nim Game Tree



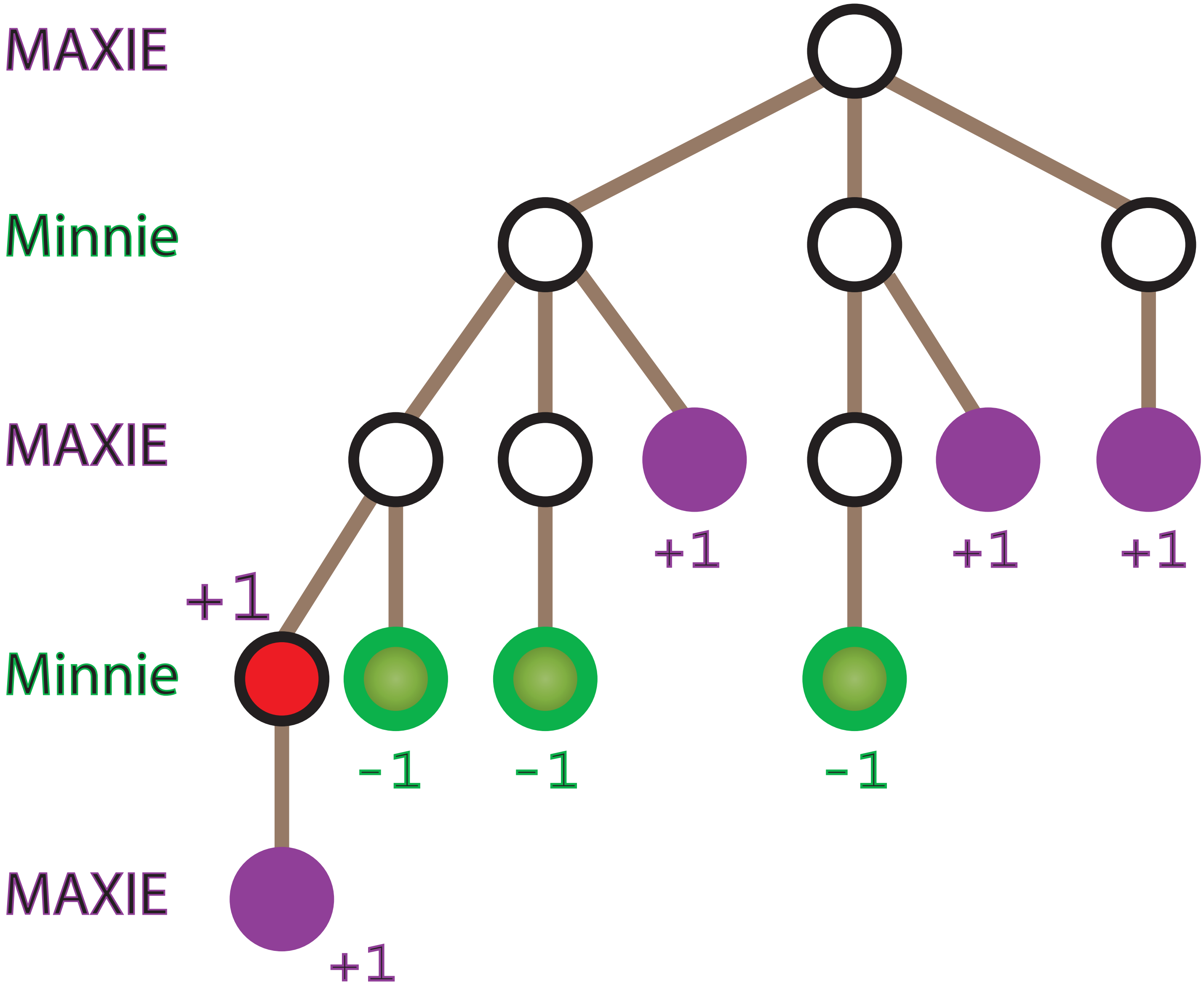
A Nim Game Tree



Nim game tree with leaf values



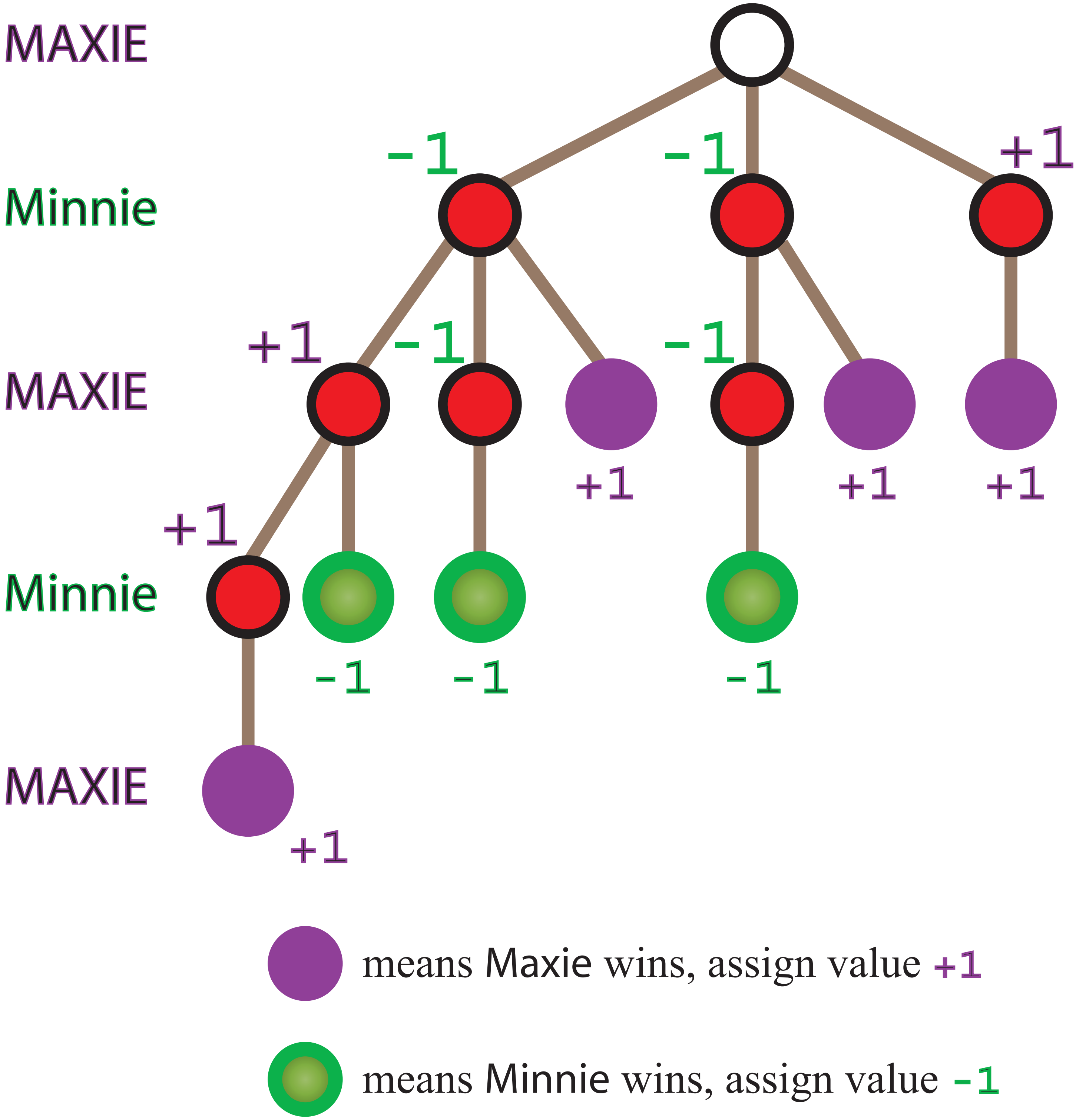
Now compute interior node values:



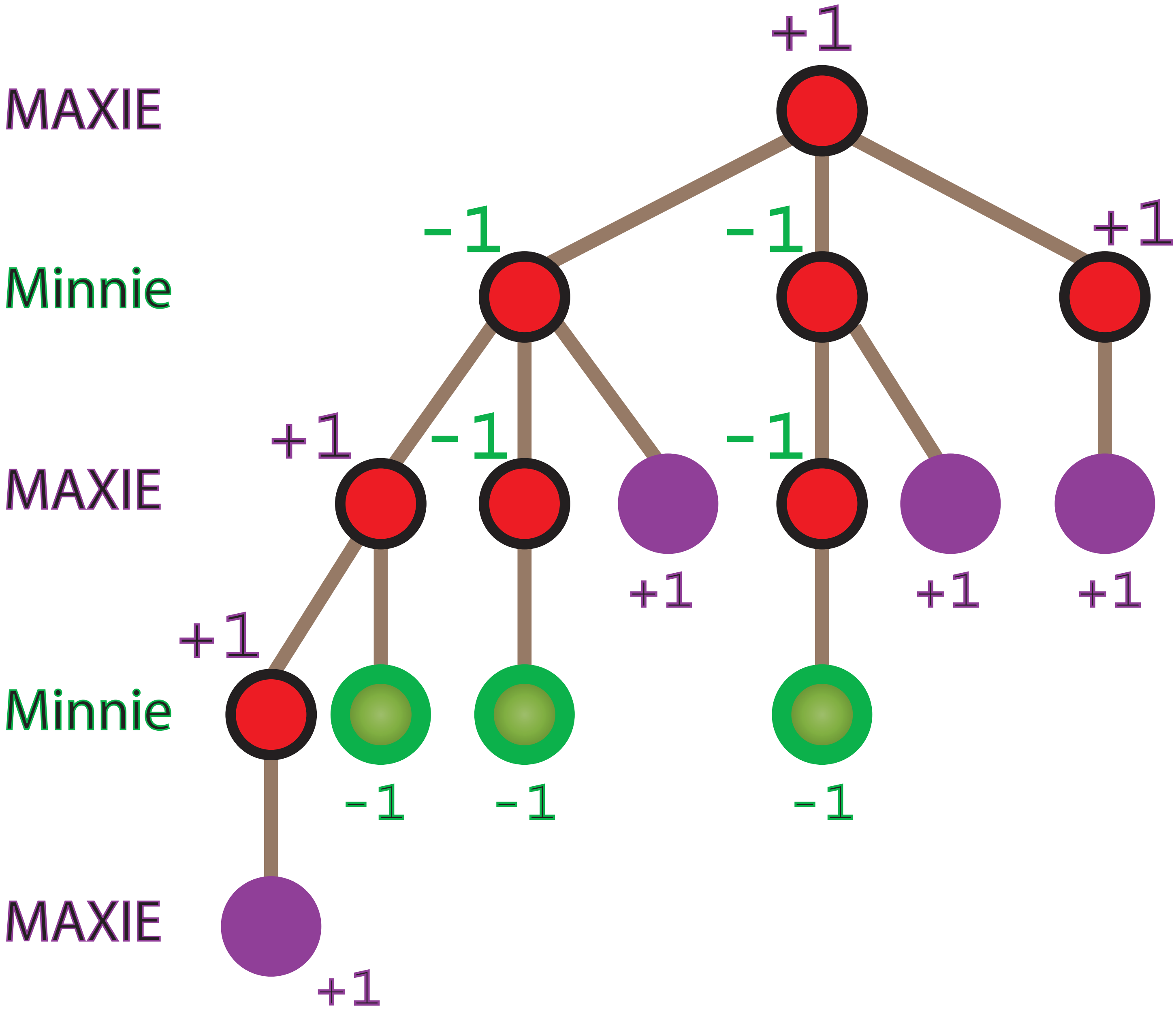
means Maxie wins, assign value +1

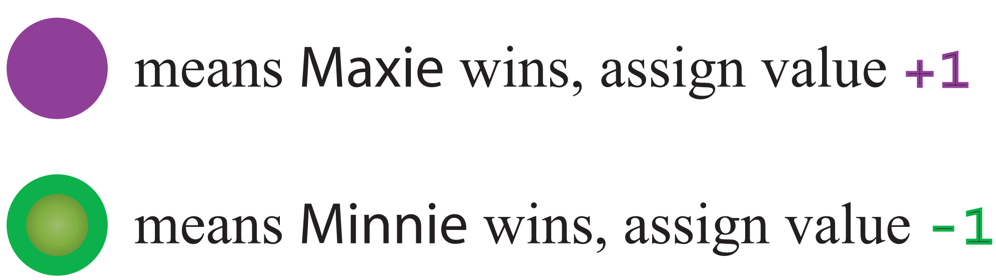
means Minnie wins, assign value -1

Now compute interior node values:

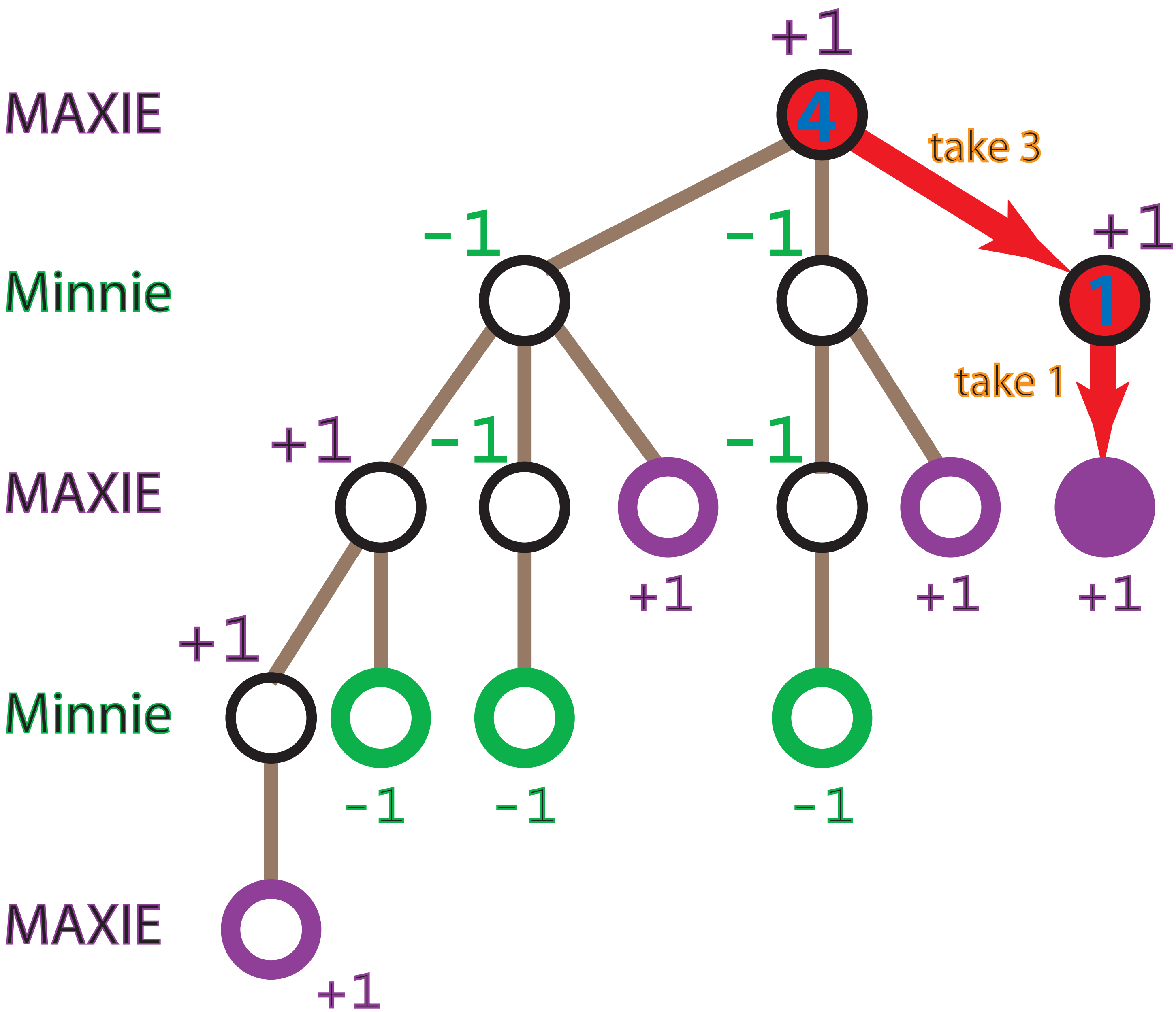


Now compute interior node values:





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 **`PLAYER`**s assume
optimal play by both **Maxie** and **Minnie** relative to the
estimator.

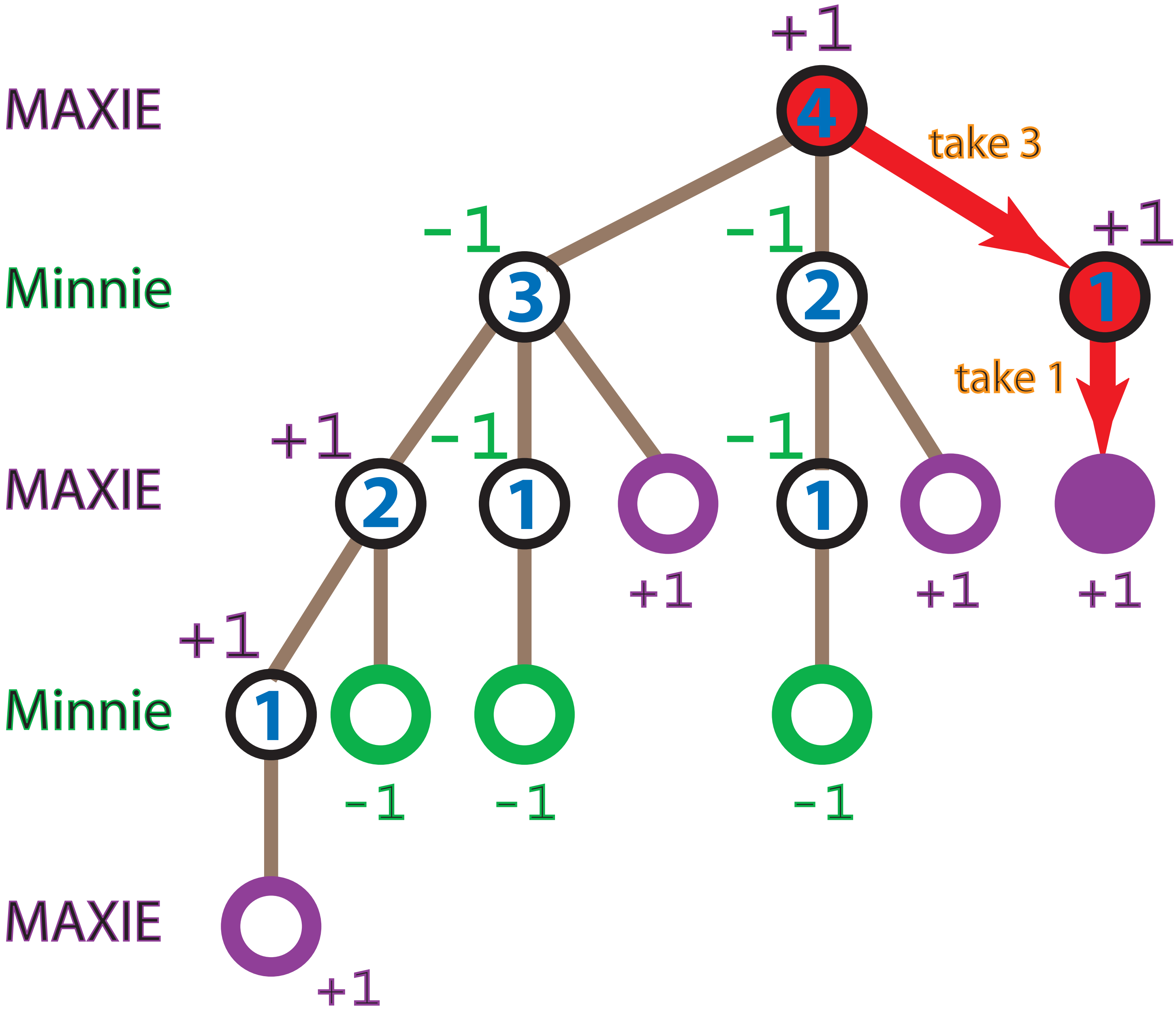
Nim has perfect estimator

Player making move can win for sure iff

$$\mathbf{n \bmod 4 \neq 1}$$

(n is number of pieces)

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		MiniMax		Referee
Connect4	GAME	AlphaBeta	PLAYER	
Checkers		Human		VerboseRef
Chess				
structures	signature	functors	signature	functors

(rough picture; there will be a few more administrative layers)

GAME Signature

```
signature GAME =
sig

end
```

GAME Signature

```
signature GAME =
sig
    datatype player = Minnie | Maxie              (* concrete *)
```

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).

```
end
```

GAME Signature

```
signature GAME =
sig
    datatype player = Minnie | Maxie                     (* concrete *)
    datatype outcome = Winner of player | Draw   (* concrete *)
```

The concrete datatype **outcome** models the idea that
either one of the players wins or there is a draw, once a game ends.

```
end
```

GAME Signature

```
signature GAME =
sig
    datatype player = Minnie | Maxie                   (* concrete *)
    datatype outcome = Winner of player | Draw         (* concrete *)
    datatype status = Over of outcome | In_play        (* concrete *)
```

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

The concrete datatype **status** models this aspect of the game.

```
end
```

GAME Signature

```
signature GAME =
sig
    datatype player = Minnie | Maxie                    (* concrete *)
    datatype outcome = Winner of player | Draw          (* concrete *)
    datatype status = Over of outcome | In_play         (* concrete *)

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

The types `state` and `move` depend on the particular game being played, so we leave them abstract in the signature.

```
end
```

GAME Signature

```
signature GAME =
sig
    datatype player = Minnie | Maxie                   (* concrete *)
    datatype outcome = Winner of player | Draw         (* concrete *)
    datatype status = Over of outcome | In_play        (* concrete *)

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

    val start : state
```

This line of the signature says that every particular game implementation must specify a value representing the start state of the game.

```
end
```

GAME Signature

```
signature GAME =
sig
    datatype player = Minnie | Maxie                    (* concrete *)
    datatype outcome = Winner of player | Draw          (* concrete *)
    datatype status = Over of outcome | In_play         (* concrete *)

    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)

```
end
```

GAME Signature

```
signature GAME =
sig
    datatype player = Minnie | Maxie                 (* concrete *)
    datatype outcome = Winner of player | Draw       (* concrete *)
    datatype status = Over of outcome | In_play      (* concrete *)

    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.)

```
end
```

GAME Signature

```
signature GAME =
sig
    datatype player = Minnie | Maxie                 (* concrete *)
    datatype outcome = Winner of player | Draw       (* concrete *)
    datatype status = Over of outcome | In_play      (* concrete *)

    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

end
```

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                (* concrete *)
    datatype outcome = Winner of player | Draw      (* concrete *)
    datatype status = Over of outcome | In_play     (* concrete *)

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

    val start : state

    val moves : state -> mov
    val make_move : state *

    val status : state -> status
    val player : state -> player
                                                    (* concrete *)
    datatype est = Definitely of outcome | Guess of int
    val estimate : state -> est

end
```

A more general approach would place the estimator in a separate module. It is here for presentational simplicity.

(REQUIRE that the state be `In_play`)

(**CAUTION**: `estimate` need not provide useful info)

GAME Signature

```
signature GAME =
sig
    datatype player = Minnie | Maxie                   (* concrete *)
    datatype outcome = Winner of player | Draw         (* concrete *)
    datatype status = Over of outcome | In_play        (* concrete *)

    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
                                                       (* concrete *)
    datatype est = Definitely of outcome | Guess of int
    val estimate : state -> est
  . . . (* functions to create string representations *)
end
```

Nim Structure

```
structure Nim : GAME =
struct



end
```

Nim Structure

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

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.

```
end
```

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
```

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
```

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

```
end
```

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)

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.)

```
end
```

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.

```
end
```

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
```

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
```

Recall that Nim has a perfect estimator (generally a game will not).

VeryDumbNim Structure

```
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
```

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

    . . .
```

We have not yet implemented the two views, so let us do that now:

```
end
```

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.

```
end
```

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.

```
end
```

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

   . . . (* functions to create string representations *)
end
```

PLAYER Signature

```
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.

Human Player

```
functor HumanPlayer (G : GAME) : PLAYER =
struct
```

The functor expects a **GAME** and returns a **PLAYER**, meaning:

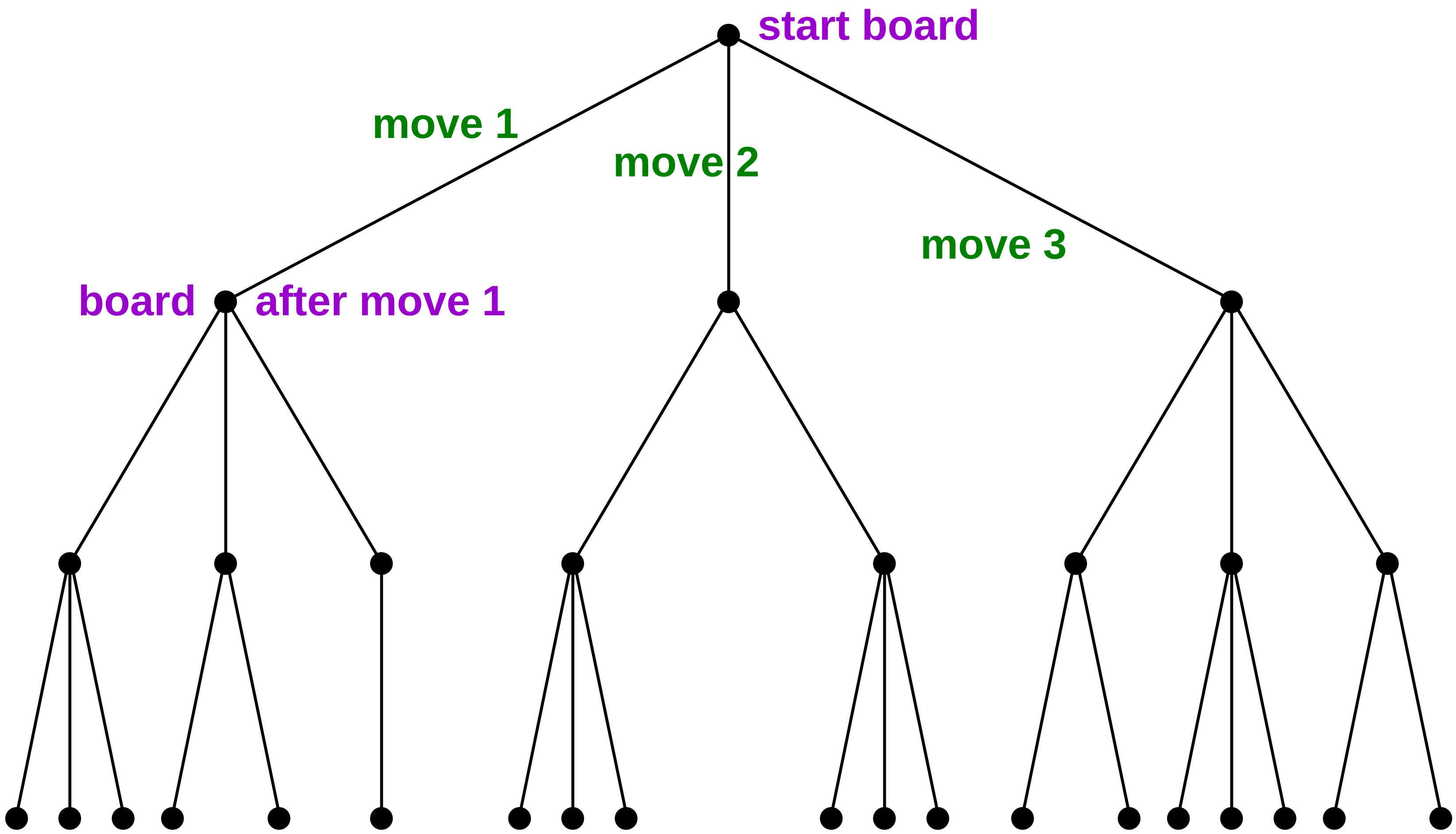
The code we write must provide a structure satisfying the **PLAYER** signature (think of that as an interface for playing games) that will work with any game **G** satisfying the **GAME** signature.

```
end
```

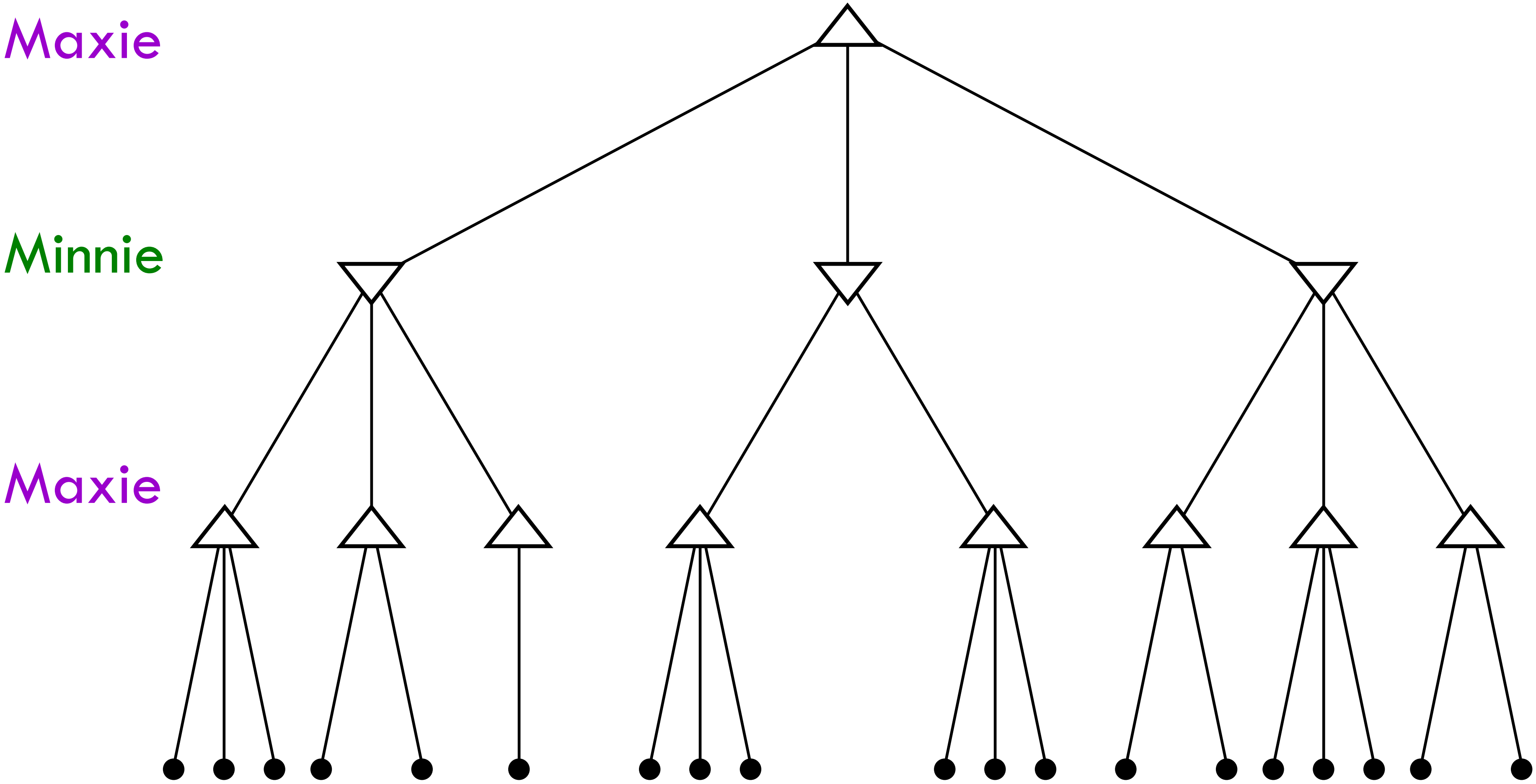
Human Player

```
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
```

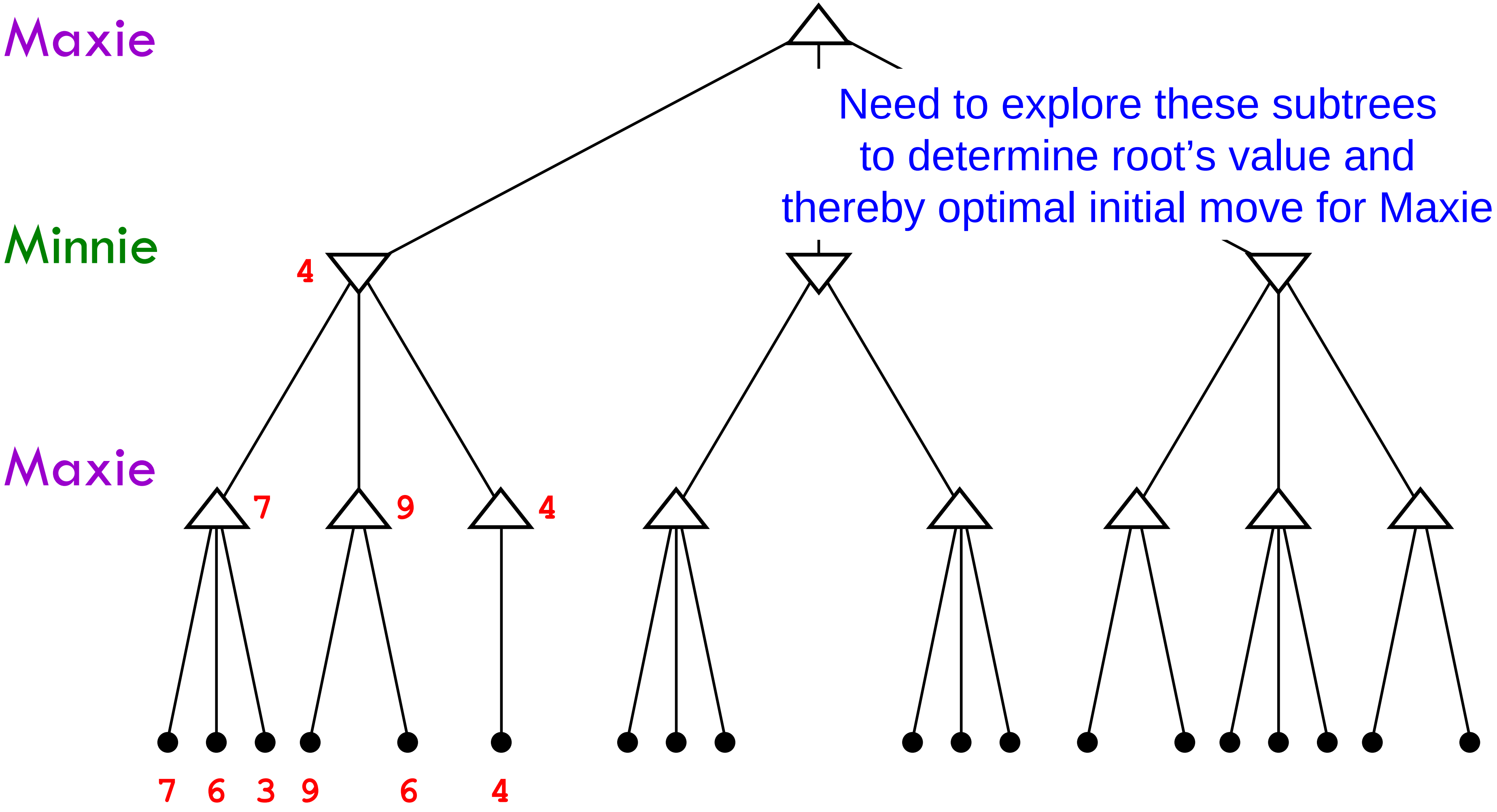
Recall: Game as tree of alternating player moves



Recall: Optimal Play from Mini-Max



Recall: Optimal Play from Mini-Max



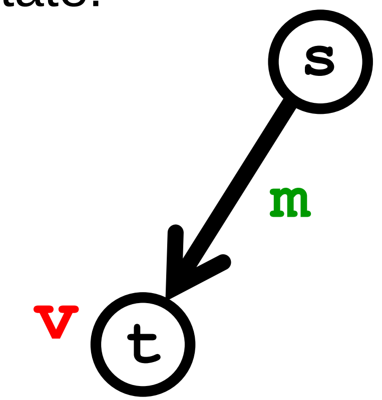
SETTINGS & PLAYER

```
signature SETTINGS =
sig
    structure Game : GAME    (* parameter *)
    val depth : int
end


signature PLAYER =
sig
    structure Game : GAME    (* parameter *)
    val next_move : Game.state -> Game.move
end
```

Functorize MiniMax Player

```
functor MiniMax (Settings : SETTINGS) : PLAYER =
struct
    structure Game = Settings.Game
    structure G = Game

    type edge = G.move * G.est
    fun emv (m,v) = m
    fun evl (m,v) = v
```

An edge represents a move from the current state, along with a value attributed to the resulting state:



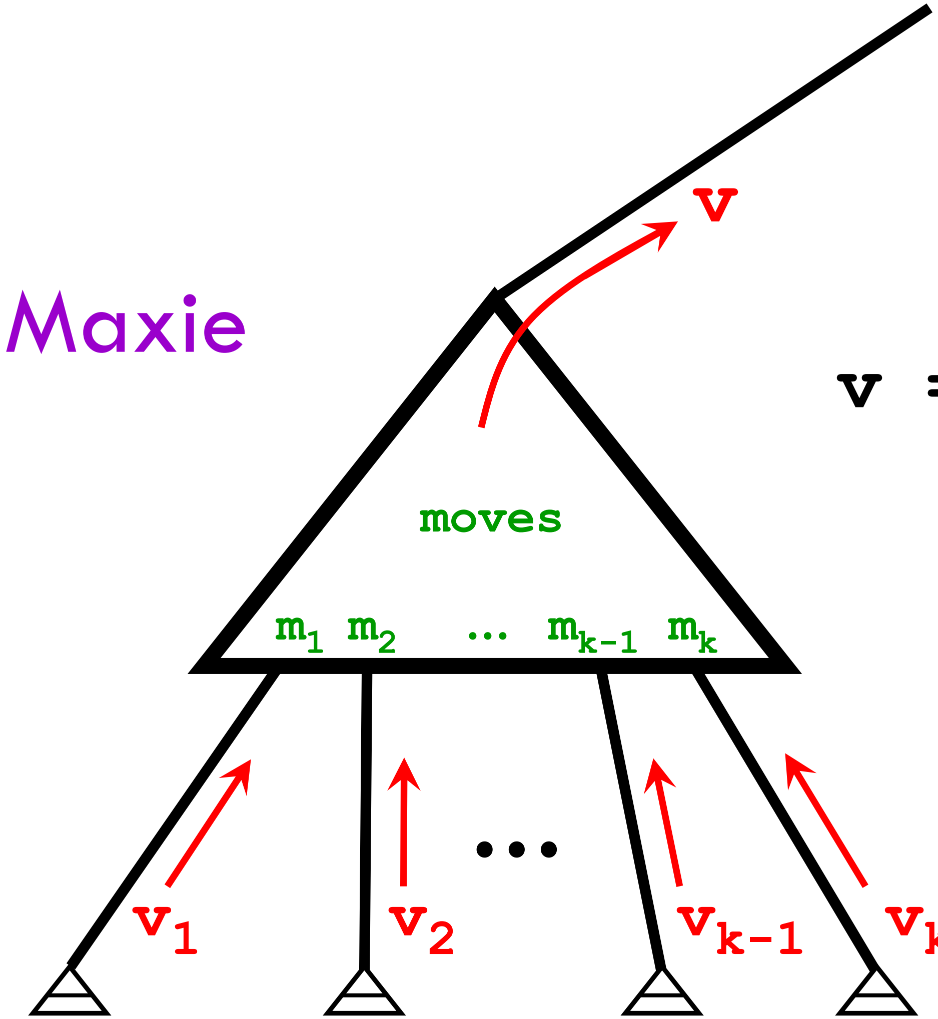

```
make_move (s,m) ≅ t
```

(`v` is `t`’s MiniMax value computed recursively)

Functorize MiniMax Player

```
functor MiniMax (Settings : SETTINGS) : PLAYER =
struct
    structure Game = Settings.Game
    structure G = Game

    type edge = G.move * G.est
    fun emv (m,v) = m
    fun evl (m,v) = v

    (* leq : G.est * G.est -> bool *)
    fun leq (x, y) =  . . .

    (* max, min : edge * edge -> edge *)
    fun max (e1, e2) = if leq (evl e2, evl e1) then e1 else e2
    fun min (e1, e2) = if leq (evl e1, evl e2) then e1 else e2

    (* choose : G.player -> edge Seq.seq -> edge *)
    fun choose G.Maxie  = Seq.reduce1 max
      | choose G.Minnie = Seq.reduce1 min
```

Mini-Max at a Maxie Node

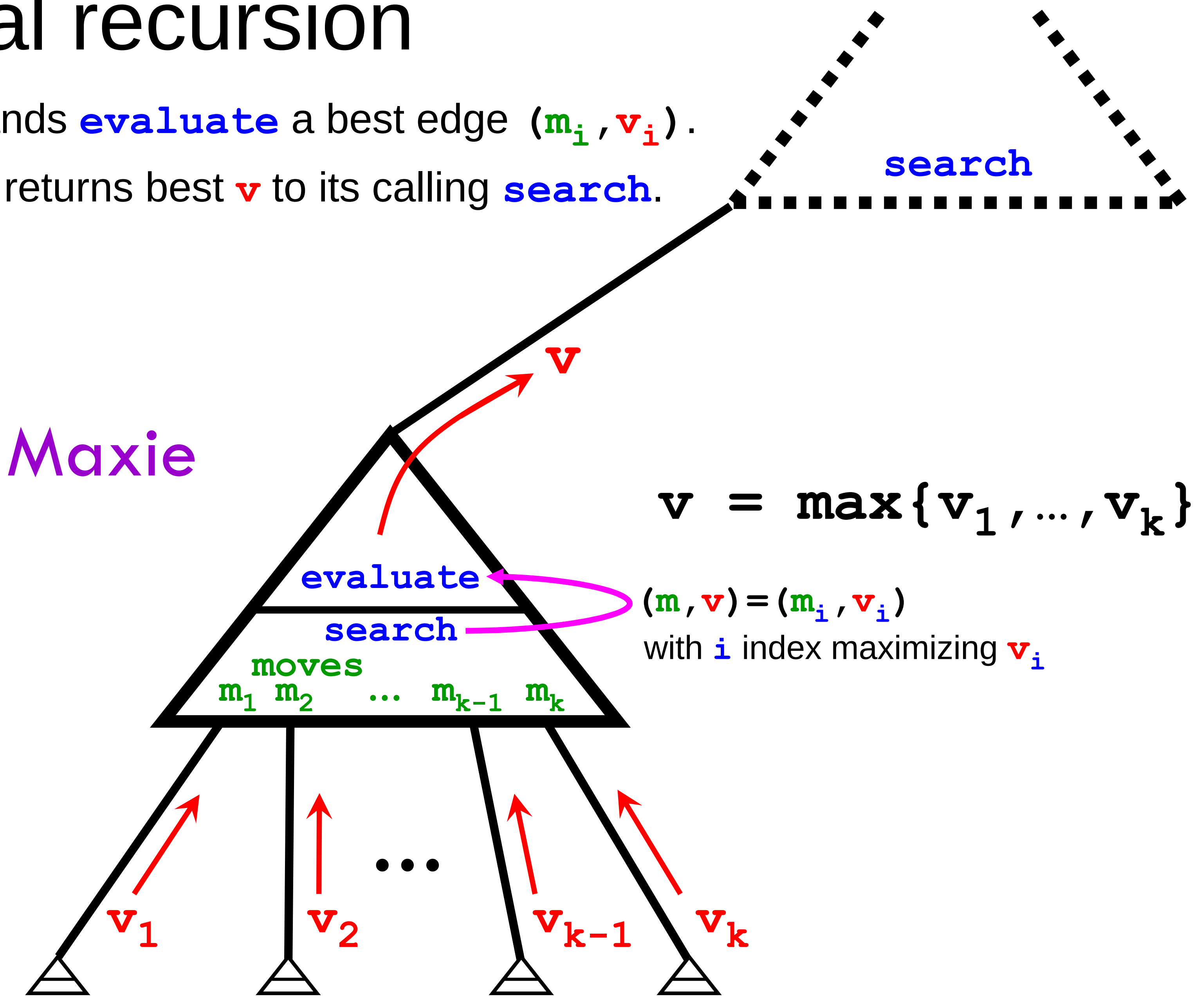


$$v = \max\{v_1, \ldots, v_k\}$$

mutual recursion

search hands **evaluate** a best edge (m_i, v_i).

evaluate returns best v to its calling **search**.



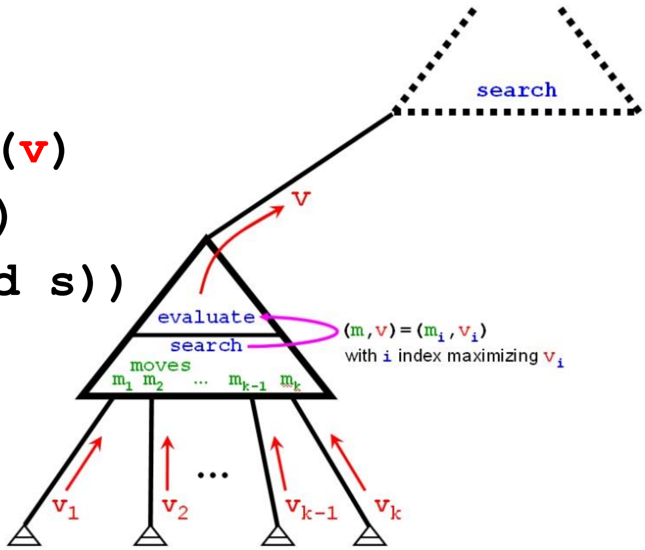
Functorize MiniMax Player (cont)

```
(* search : int -> G.state -> edge                      *)
(* REQUIRES: depth d > 0 and G.status(s) == In_play. *)
fun search d s =
     choose (G.player s)
            (Seq.map
              (fn m => (m, evaluate (d-1) (G.make_move(s,m))))
              (G.moves s))

(* evaluate : int -> G.state -> G.est  *)
(* REQUIRES : d ≥ 0.                   *)
and evaluate d s =
     (case (G.status s, d) of
          (G.Over(v), _) => G.Definitely(v)
        | (G.In_play, 0) => G.estimate(s)
        | (G.In_play, _) => evl (search d s))
```

Functorize MiniMax Player (cont)

```
(* search : int -> G.state -> edge                        *)
(* REQUIRES: depth d > 0 and G.status(s) == In_play.  *)
fun search d s =
     choose (G.player s)
            (Seq.map
              (fn m => (m, evaluate (d-1) (G.make_move(s,m))))
              (G.moves s))

(* evaluate : int -> G.state -> G.est  *)
(* REQUIRES : d ≥ 0.                    *)
and evaluate d s =
     (case (G.status s, d) of
           (G.Over(v), _) => G.Definitely(v)
         | (G.In_play, 0) => G.estimate(s)
         | (G.In_play, _) => evl (search d s))
```

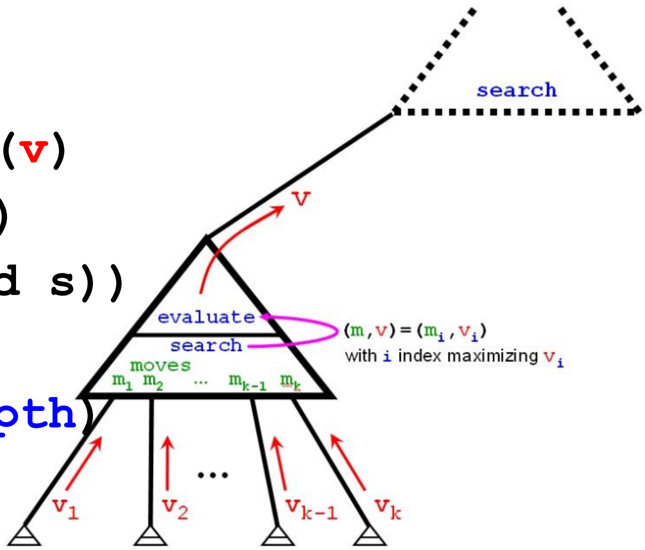
select the value of the best edge



Functorize MiniMax Player (cont)

```
(* search : int -> G.state -> edge                        *)
(* REQUIRES: depth d > 0 and G.status(s) == In_play.  *)
fun search d s =
     choose (G.player s)
            (Seq.map
              (fn m => (m, evaluate (d-1) (G.make_move(s,m))))
              (G.moves s))

(* evaluate : int -> G.state -> G.est  *)
(* REQUIRES : d ≥ 0.                   *)
and evaluate d s =
     (case (G.status s, d) of
           (G.Over(v), _) => G.Definitely(v)
         | (G.In_play, 0) => G.estimate(s)
         | (G.In_play, _) => evl (search d s))

val next_move = emv o (search Settings.depth)
```

select the move from the best edge



Functorize MiniMax Player (cont)

```
(* search : int -> G.state -> edge                     *)
(* REQUIRES: depth d > 0 and G.status(s) == In_play. *)
fun search d s =
     choose (G.player s)
            (Seq.map
              (fn m => (m, evaluate (d-1) (G.make_move(s,m))))
              (G.moves s))

(* evaluate : int -> G.state -> G.est  *)
(* REQUIRES : d ≥ 0.                   *)
and evaluate d s =
     (case (G.status s, d) of
           (G.Over(v), _) => G.Definitely(v)
         | (G.In_play, 0) => G.estimate(s)
         | (G.In_play, _) => evl (search d s))
```

This is *the* function specified in the PLAYER signature, accessible to the outside world.

```
val next_move = emv o (search Settings.depth)
```

Functorize MiniMax Player (cont)

```
(* search : int -> G.state -> edge                      *)
(* REQUIRES: depth d > 0 and G.status(s) == In_play. *)
fun search d s =
     choose (G.player s)
            (Seq.map
              (fn m => (m, evaluate (d-1) (G.make_move(s,m))))
              (G.moves s))

(* evaluate : int -> G.state -> G.est  *)
(* REQUIRES : d ≥ 0.                    *)
and evaluate d s =
     (case (G.status s, d) of
           (G.Over(v), _) => G.Definitely(v)
         | (G.In_play, 0) => G.estimate(s)
         | (G.In_play, _) => evl (search d s))

val next_move = emv o (search Settings.depth)

end (* functor MiniMax *)
```

TWO_PLAYERS & GO

```
signature TWO_PLAYERS =
sig
    structure Maxie  : PLAYER   (* parameter *)
    structure Minnie : PLAYER   (* parameter *)
    sharing Maxie.Game = Minnie.Game
end



signature GO =
sig
    val go : unit -> unit
end
```

Functorize Playing, using a Referee

```
functor Referee (P : TWO_PLAYERS) : GO =
struct
    structure G = P.Maxie.Game
    structure H = P.Minnie.Game

    (* run : G.state -> string *)
    fun run s =
        (case (G.status s, G.player s) of
             (G.Over(v), _) => G.outcome_to_string(v)
           | (G.In_play, G.Maxie) =>
                 run (G.make_move (s, P.Maxie.next_move s))
           | (G.In_play, G.Minnie) =>
                 run (H.make_move (s, P.Minnie.next_move s)))

    fun go () = print (run (G.start) ^ "\n")
end
```

Human vs depth-3 MiniMax for Nim

```
structure NimHuman = HumanPlayer(Nim)     (* Nim : GAME *)

structure NimSet3 : SETTINGS =
struct
   structure Game = Nim
   val depth = 3
end

structure Nim3MM = MiniMax(NimSet3)

structure HvM : TWO_PLAYERS =
struct
   structure Maxie = NimHuman
   structure Minnie = Nim3MM
end

structure Nim_RefHvM = Referee(HvM)

Nim_RefHvM.go()
```

That is all.