Tic-tac-toe

Lars-Henrik Eriksson

Functional Programming 1

Original presentation by Tjark Weber

Take-Home Exam

Take-Home Exam

If you have passed all lab assignments, you will pass the course with a grade of (at least) 3. The exam is optional.

The exam will be an **individual take-home** exam. You do **not** need to register for the exam. You can only take the exam once.

The exam instructions are already available on the Student Portal. The submission deadline will be on October 27.

An average student should expect to spend roughly 40 hours on the exam (more if you are ambitious). **Plan your time accordingly!** Contact me in case of scheduling issues due to sickness etc.

The SML Basis Library

The SML Basis Library: Introduction

We have been using a number of "built-in" types (e.g., `int`, `bool`, `list`), functions (e.g., **op**+, `print`) and values (e.g., `true`, `[]`).

So, what exactly is available?

The SML Basis Library: Introduction

We have been using a number of "built-in" types (e.g., `int`, `bool`, `list`), functions (e.g., **op**+, `print`) and values (e.g., `true`, `[]`).

So, what exactly is available?

The SML Basis Library: Description

The SML Basis Library provides interfaces and operations for basic types, such as integers and strings, support for input and output (I/O), interfaces to basic operating system interfaces, and support for standard datatypes, such as options and lists. The Library does not attempt to define higher-level APIs, such as collection types or graphical user-interface components. These APIs are left for other libraries.

`http://sml-family.org/Basis/`

The SML Basis Library: Contents

Contents:

- Basic types (i.e., bool, int, word and real)
- Standard datatypes (i.e., option and list)
- Vectors and arrays
- Text processing (e.g., char and string)
- System interfaces (e.g., files, process control)
- Sockets

The SML Basis Library is organized into structures. Most identifiers are available in some structure. Only the most frequently used identifiers are also available unqualified, in the top-level environment.

The SML Basis Library: Documentation

See `http://sml-family.org/Basis/` for a detailed list of all structures and their components.

Alternatively, to quickly see the components of a (known) structure, you can interactively **open** the structure. For example:

```
open List;

val @ = fn: 'a list * 'a list -> 'a list
exception Empty
val all = fn: ('a -> bool) -> 'a list -> bool
val app = fn: ('a -> unit) -> 'a list -> unit
...
```

The SML Basis Library: The Bigger Picture

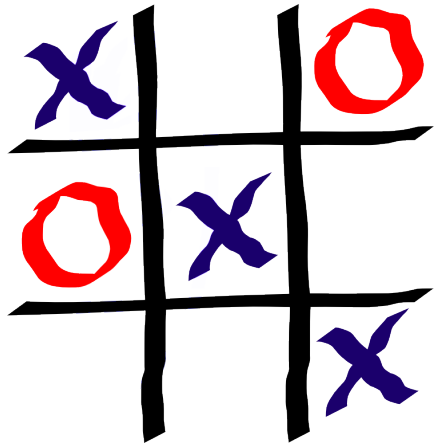
Remember that Poly/ML is just one of many different implementations (compilers) for SML.

The SML Basis Library is available on (nearly) all SML implementations. Thus, it facilitates the development of **portable** SML code.

Individual compilers may provide additional functionality. (For instance, Poly/ML provides a structure—incidentally called PolyML—with components that facilitate debugging, profiling, etc.)

Tic-tac-toe

Tic-tac-toe: Rules



1. The game is played on a 3-by-3 board that is initially empty.
2. Two players, called X and O, take turns. X goes first.
3. A player's move consists of marking an empty field with his name.
4. The first player to get three markers in a (horizontal, vertical or diagonal) row wins the game.
5. Otherwise, the game is a draw after nine turns.

Tic-tac-toe: Program

In the Student Portal you will find a complete ML program that plays tic-tac-toe.

Here we will cover the basic ideas.

Tic-tac-toe: Basic SML Types

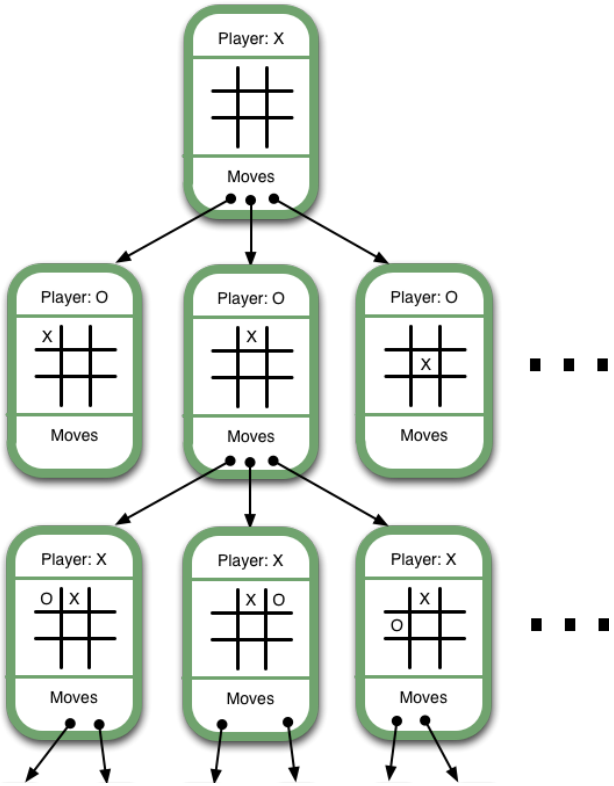
```
datatype player = X | O

type field = player option

datatype board = Board of field * field * field *
                          field * field * field *
                          field * field * field

datatype position = Position of player * board

datatype move = One | Two | Three | Four | Five
              | Six | Seven | Eight | Nine
```

Game Trees

A **game tree** is a directed graph whose nodes are positions in a game and whose edges are moves.

The complete game tree for a game is the game tree starting at the initial position and containing all possible moves from each position.

http://en.wikipedia.org/wiki/Game_tree



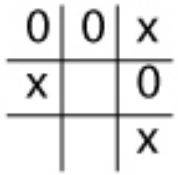
The Minimax Algorithm

The minimax algorithm is a recursive algorithm for choosing the next move in a game with two (or more) players. It works backwards, from the final positions towards the current position.

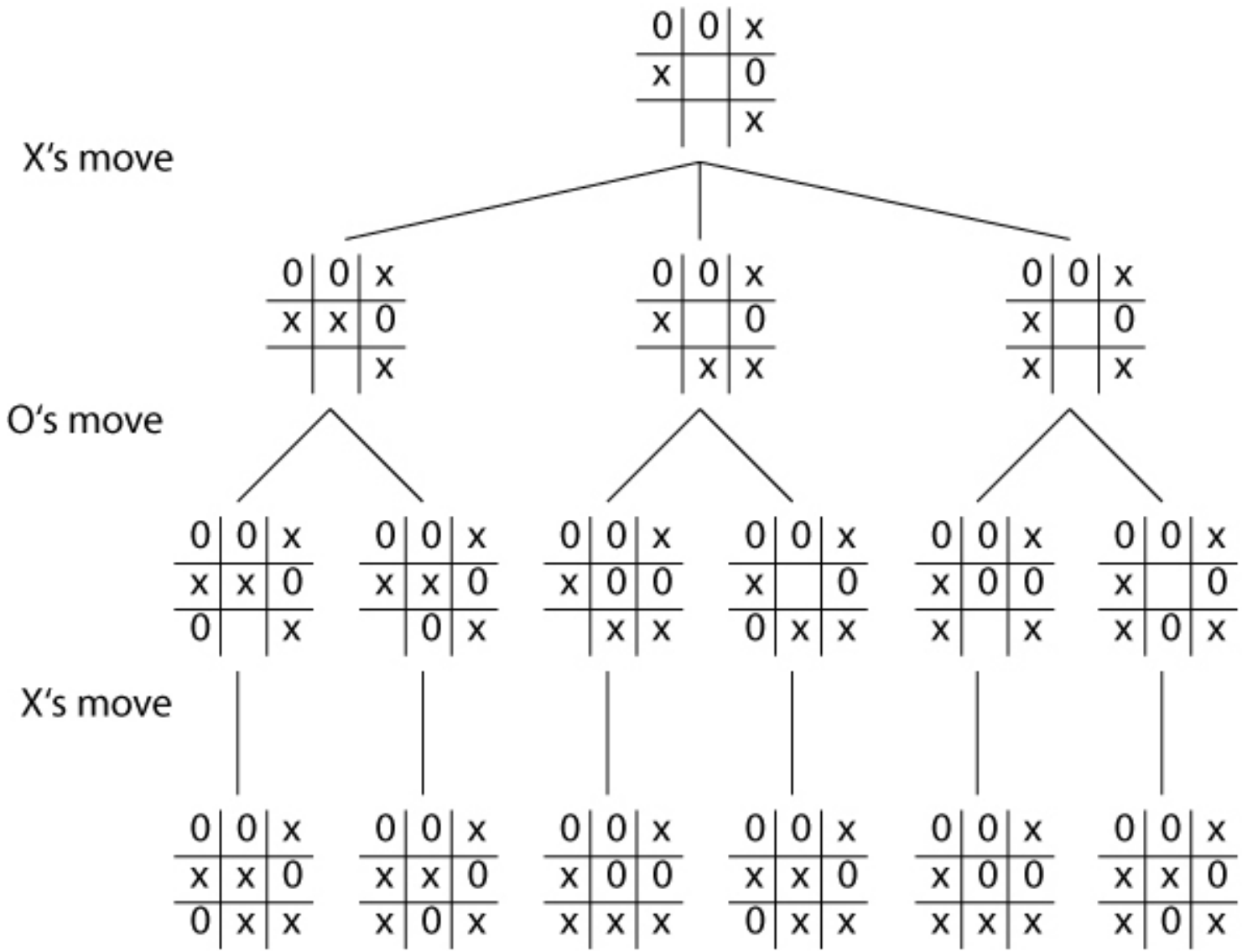
1. Expand the complete game sub-tree below the current position.
2. Determine the value (i.e., O wins, draw, X wins) of each final position.
3. Propagate these values backwards, under the assumption that each player will choose the move that's best for him.

The Minimax Algorithm: Example

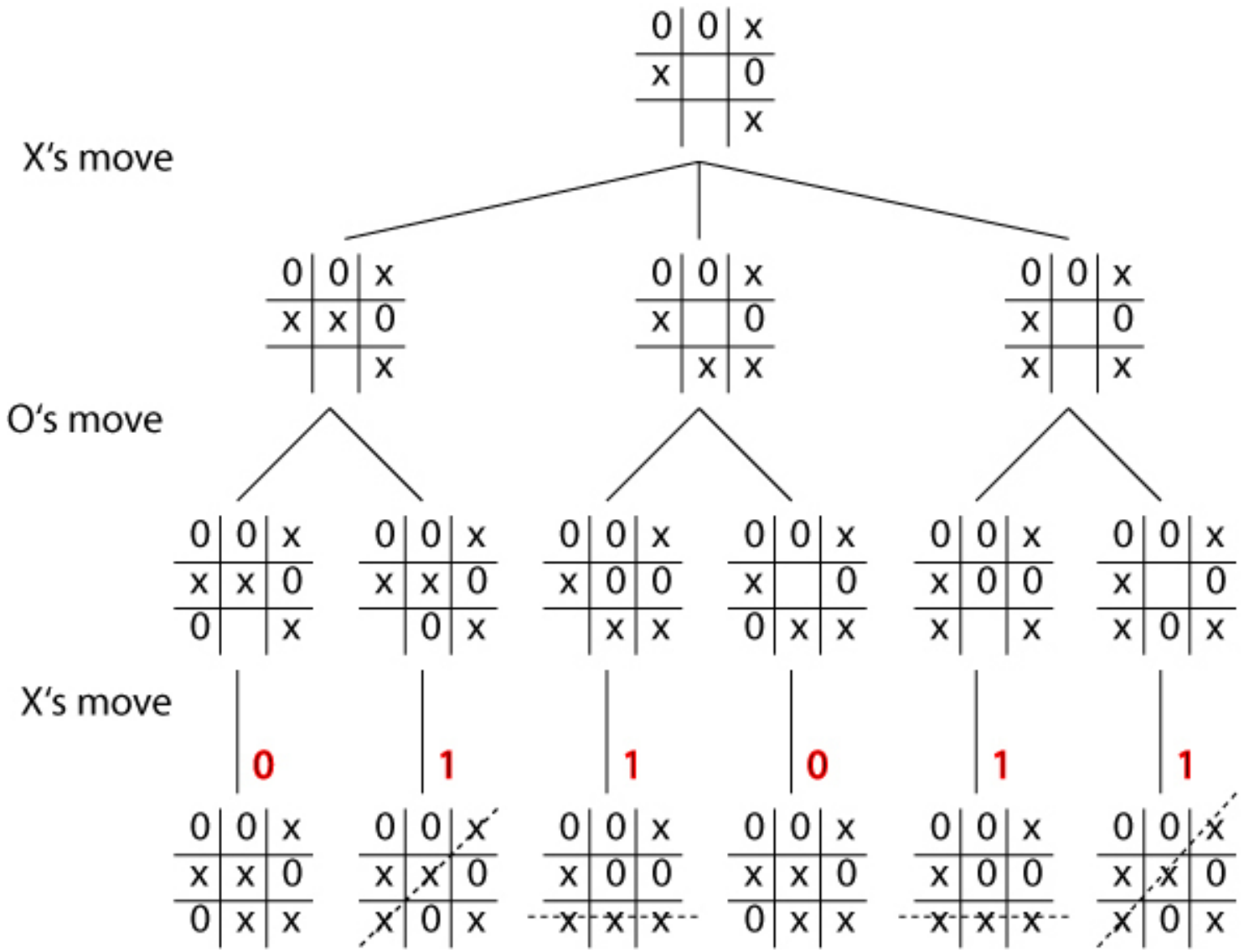
X's move



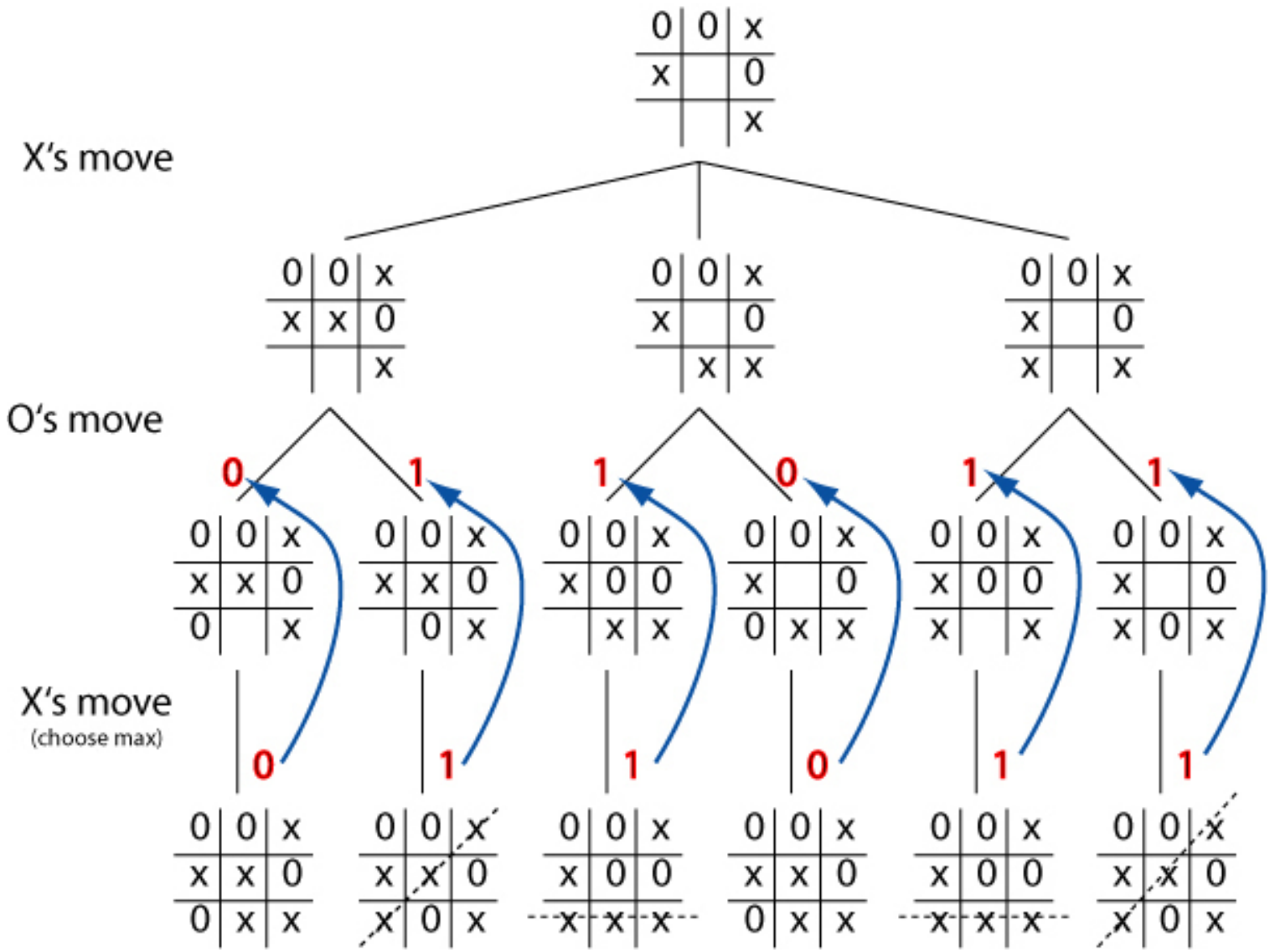
The Minimax Algorithm: Example



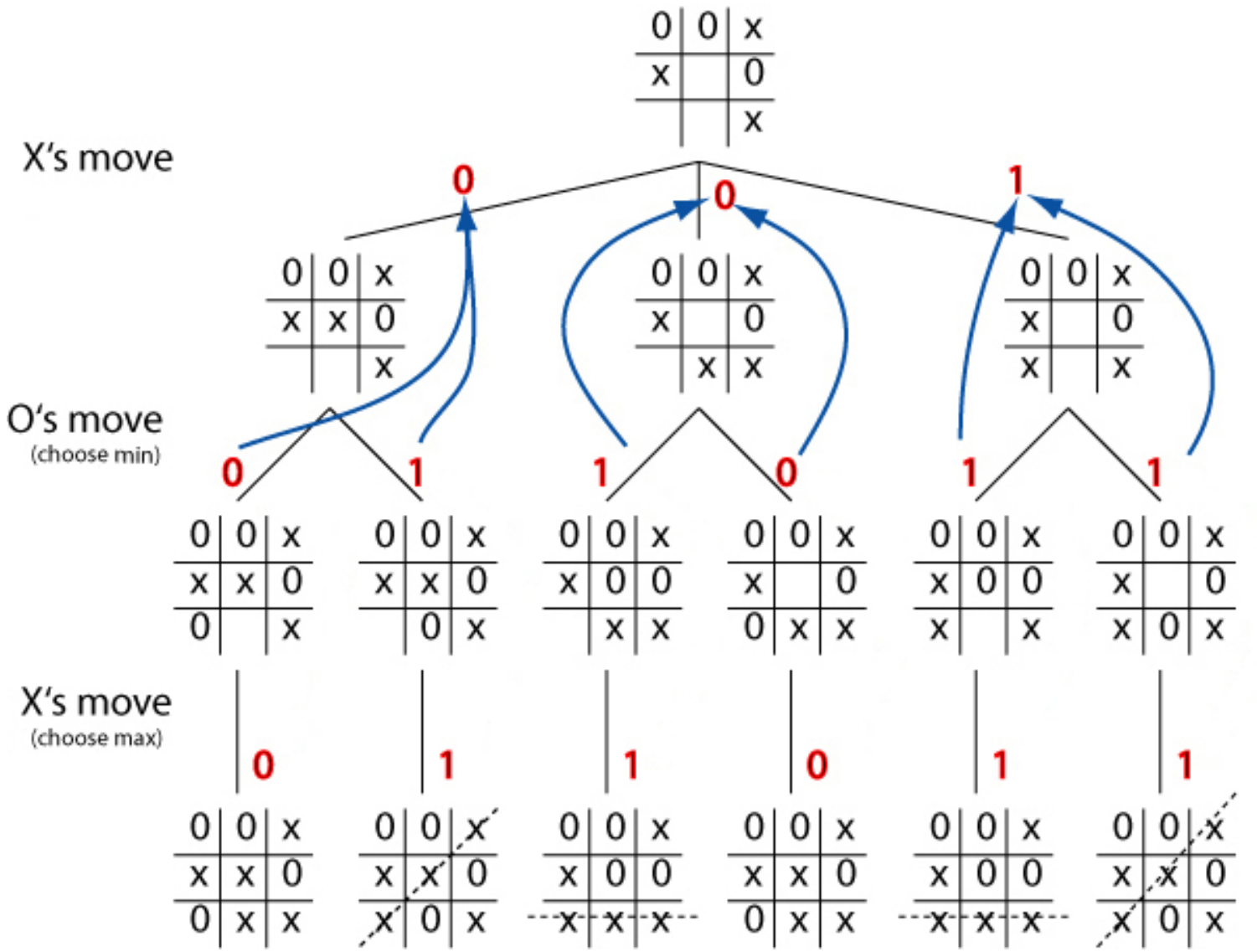
The Minimax Algorithm: Example



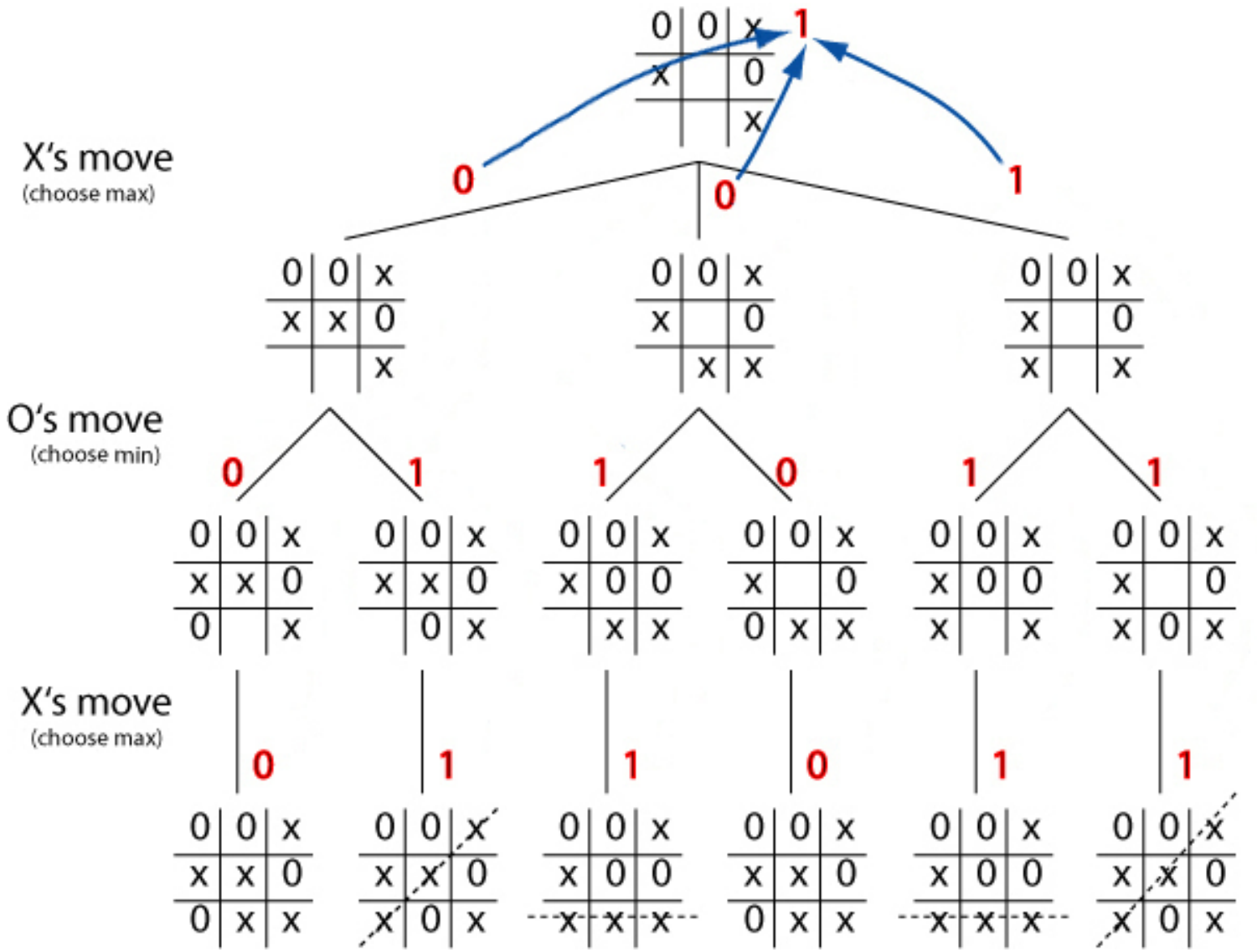
The Minimax Algorithm: Example



The Minimax Algorithm: Example



The Minimax Algorithm: Example



The Minimax Algorithm: Ingredients

To build the complete game tree, we need

- a type of game positions (board, active player, etc.),
- a type of moves (i.e., choices that a player has),
- the initial position,
- a function to enumerate all valid moves for a given position,
- a function to generate the new position obtained after a valid move.

To employ the minimax algorithm, we also need

- an evaluation function for final game positions.

The Minimax Algorithm: Code

```
fun minimax position =
let
  val moves = moves_of position
in
  if null moves then
    value_of position (* X wins > draw > O wins *)
  else
    let
      val positions = map (make_move position) moves
    in
      (if player_of position = X then max else min)
        (map minimax positions)
    end
end
```

Minimax With Limited Depth

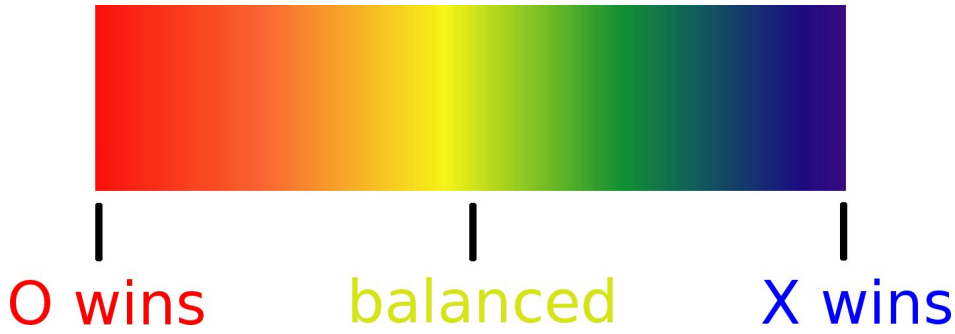
For games that are significantly more complex than Tic-tac-toe (e.g., chess, Go), it is not feasible to generate the complete game tree.

Instead, the game tree is only explored to a certain depth (starting from the current position), i.e., only a certain number of moves ahead.

The minimax algorithm can still be used, but it now requires an **evaluation heuristic** that can be applied to non-final positions.

Evaluation Heuristics

Evaluation heuristics estimate how much of an advantage players have in a given position.



They rely heavily on game-specific knowledge. Finding good evaluation heuristics is difficult.

Evaluation heuristics strike a balance between an accurate estimate and a computationally cheap estimate. (In particular, they are much cheaper than expanding the complete game sub-tree below the given position.)

Thank You!

Thank You!

There is still a lot of work ahead (Assignment 4, the exam) . . . but today was our last proper lecture. (There will also be a guest lecture about practical use of functional programming.)

So, thank you for taking this course and sticking with it until now!

There will (of course) be a **course evaluation** on the Student Portal in a few weeks' time. Please take the time to give (positive and negative) feedback. Next year's students will benefit from it!

Advanced Functional Programming

We hope that you got a first glimpse of functional programming in this course.

If you are interested, there will be a course on **Advanced Functional Programming** (5 hp) next period, covering multiple languages and more advanced concepts.