Programming Paradigms

Lecture 9. Wholemeal programming. Typed FP in other languages

Outline

- Wholemeal programming
- Why types and ADTs in particular are important?
- ADTs in other languages

Wholemeal programming

Functional Pearl: La Tour D'Hanoï

Ralf Hinze
Computing Laboratory, University of Oxford, Wolfson Building, Parks Road, Oxford, OX1 3QD, England
ralf.hinze@comlab.ox.ac.uk

Functional languages excel at *wholemeal programming*, a term coined by Geraint Jones. Wholemeal programming means to think big: work with an entire list, rather than a sequence of elements; develop a solution space, rather than an individual solution; imagine a graph, rather than a single path. The wholemeal approach often offers new insights or provides new perspectives on a given problem. It is nicely complemented by the idea of *projective programming*: first solve a more general problem, then extract the interesting bits and pieces by transforming the general program into more specialised ones. This pearl aims to demonstrate the techniques using the popular Towers of Hanoi puzzle as a running example. This puzzle has its own beauty, which we hope to expose along the way.

http://www.cs.ox.ac.uk/ralf.hinze/publications/ICFP09.pdf

Wholemeal programming

Compare

```
result = 0;
for (int i = 0; i < len(values); i++) {
  result = result + values[i] * values[i];
}
```

Wholemeal programming

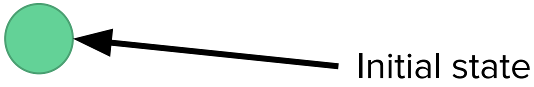
Compare

```
result = 0;
for (int i = 0; i < len(values); i++) {
  result = result + values[i] * values[i];
}
```

versus

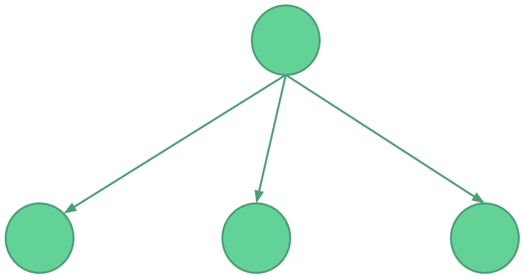
```
result = sum (map (\x -> x * x)) values
```

Example: outline of a minimax algorithm

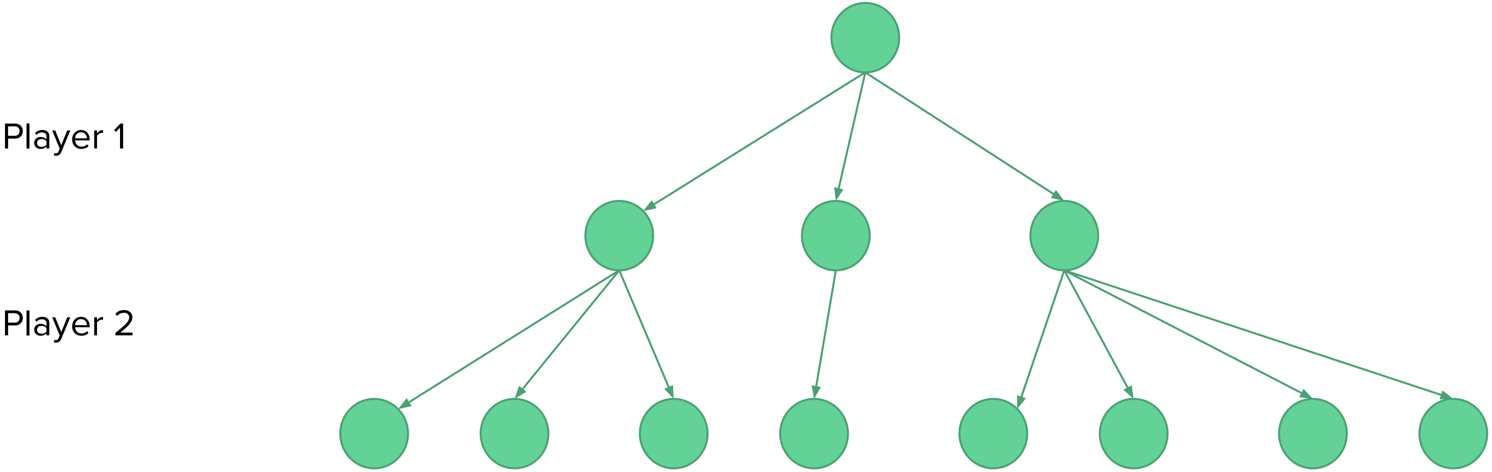


Example: outline of a minimax algorithm

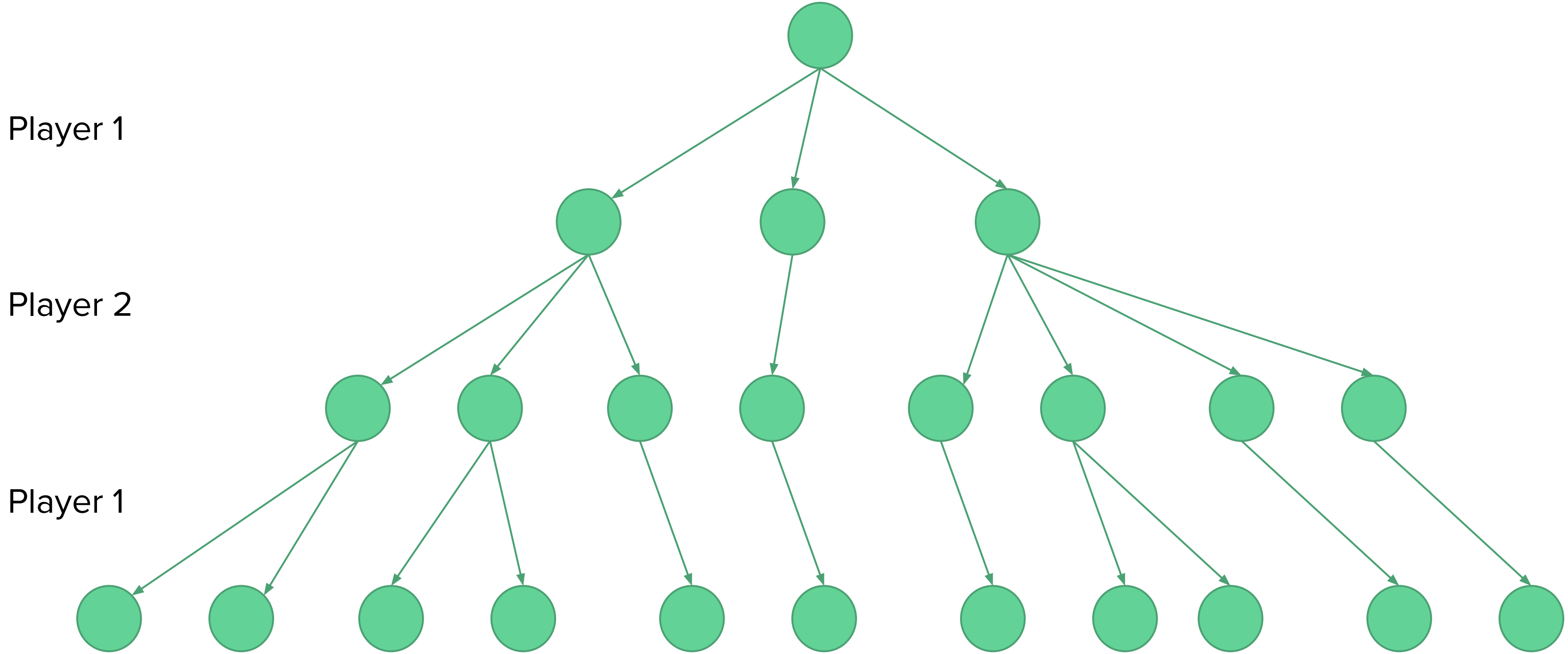
Player 1



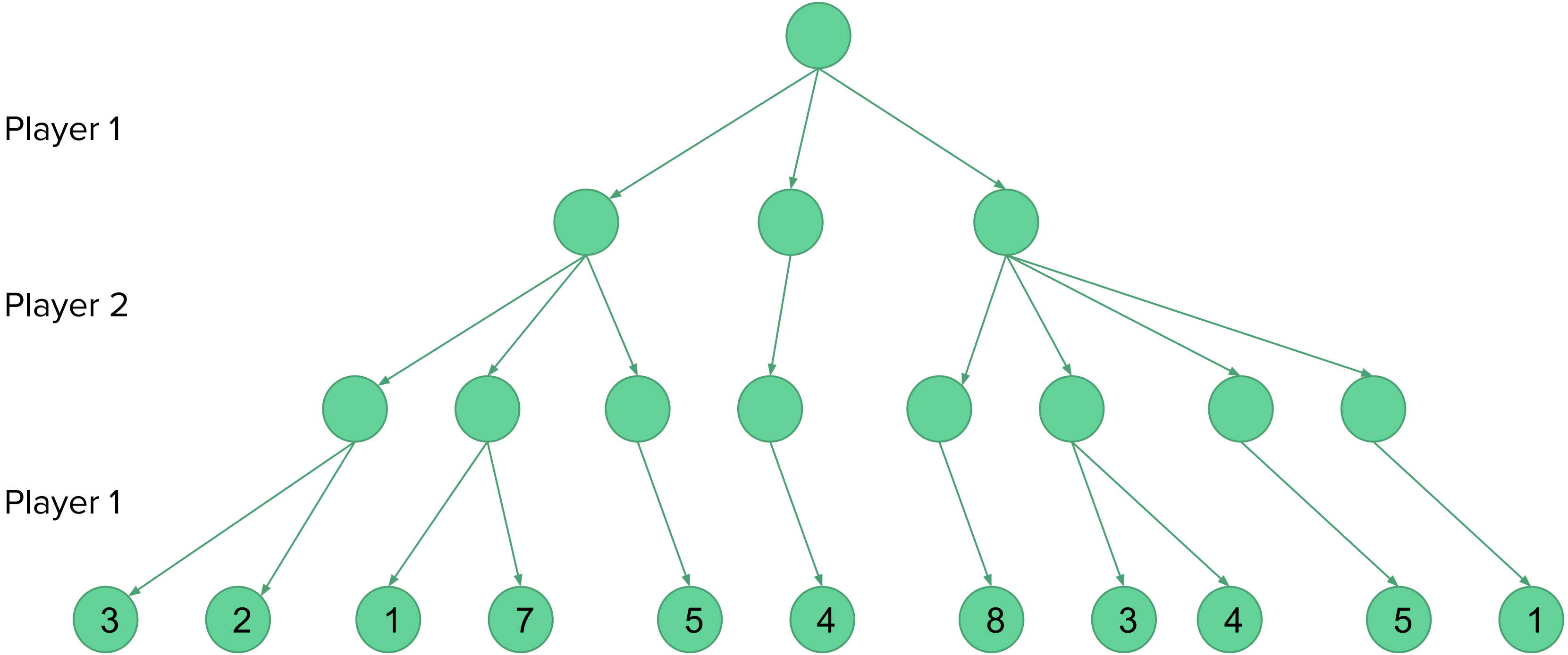
Example: outline of a minimax algorithm



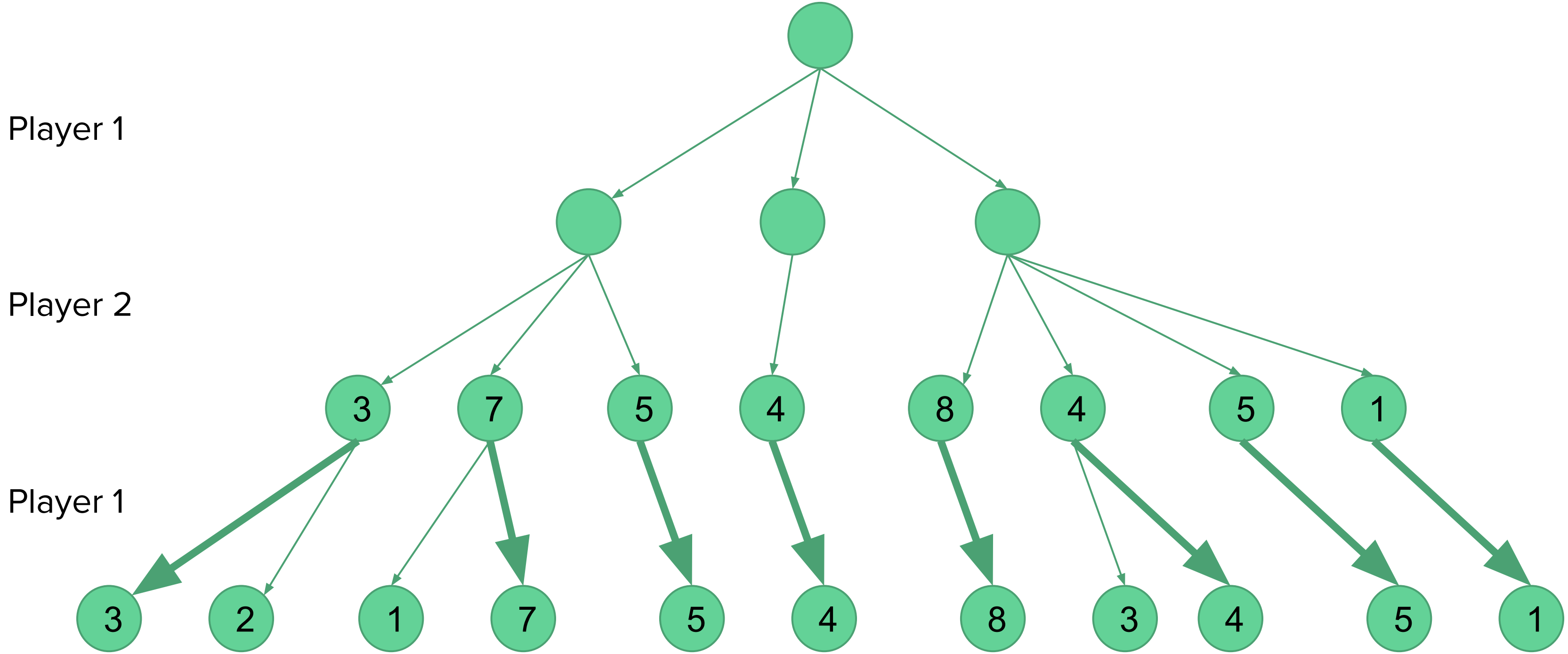
Example: outline of a minimax algorithm



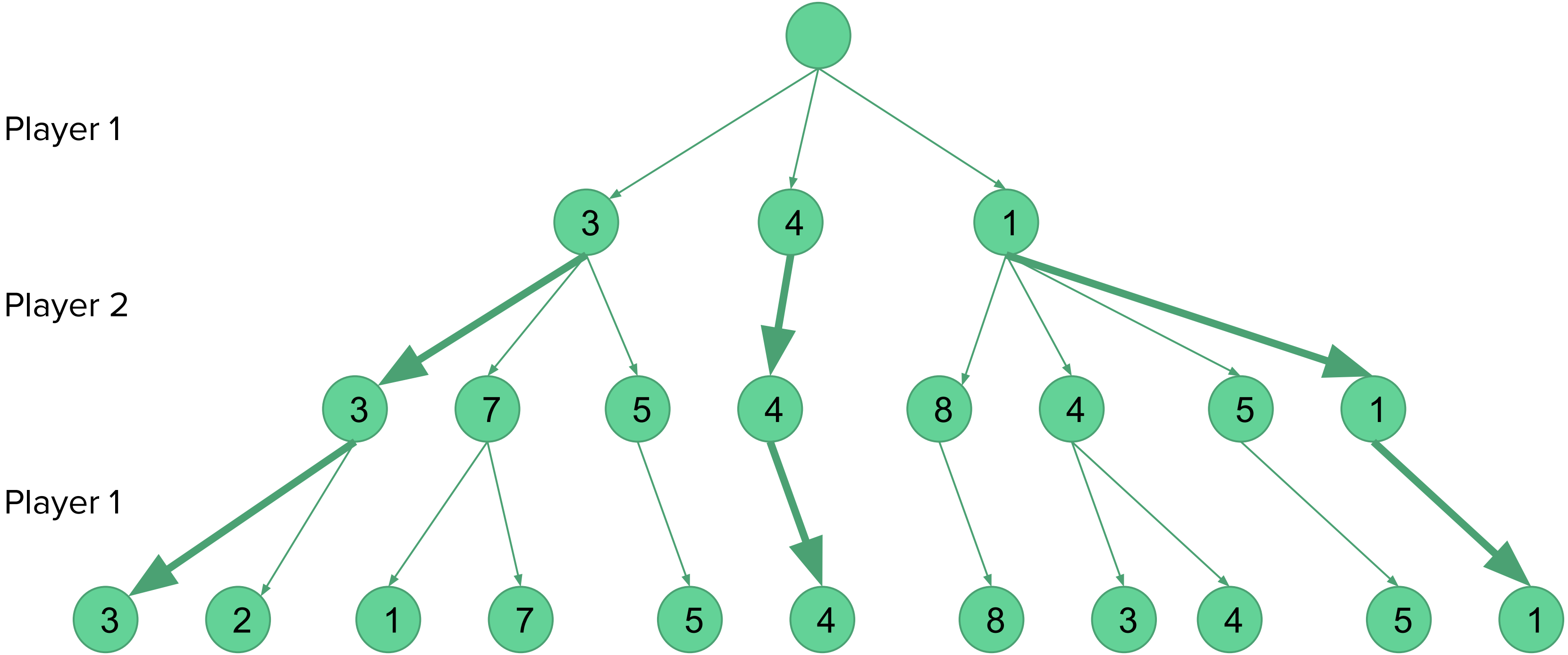
Example: outline of a minimax algorithm



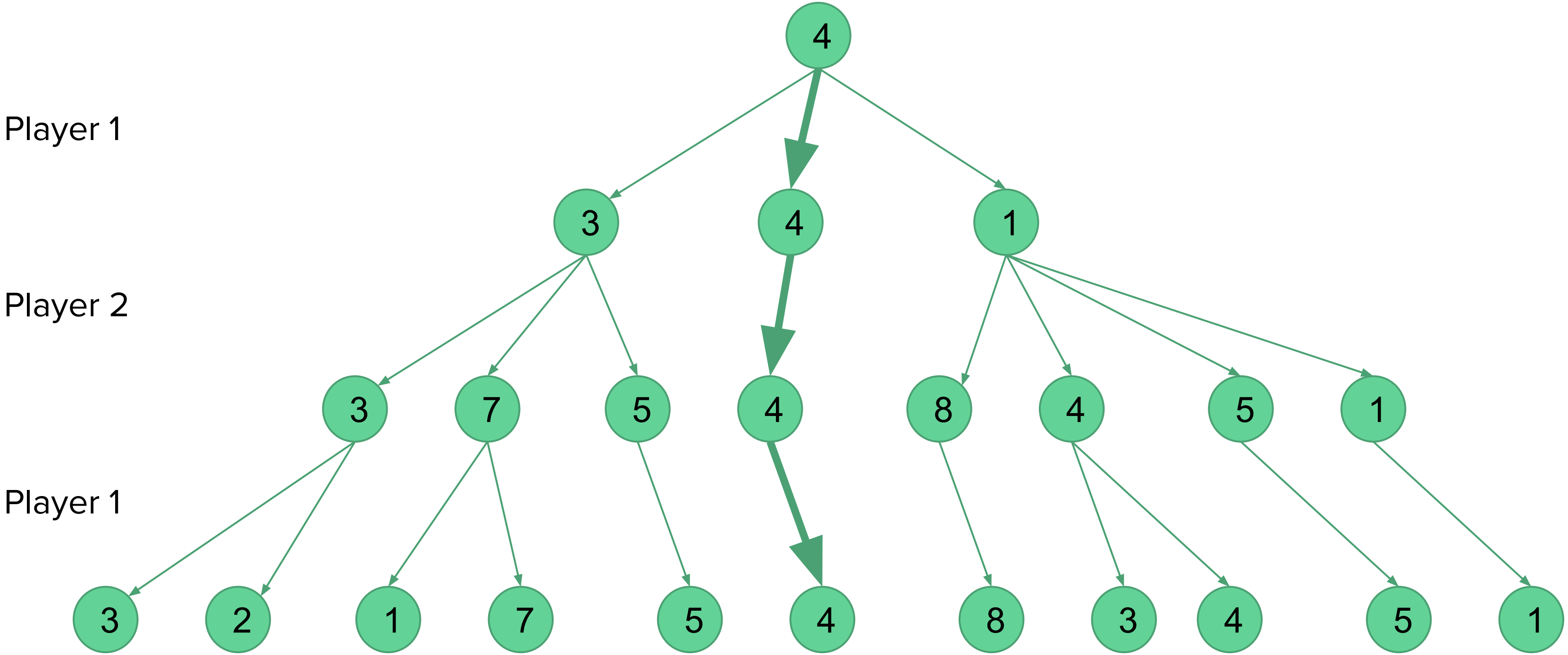
Example: outline of a minimax algorithm



Example: outline of a minimax algorithm



Example: outline of a minimax algorithm



Top-down solution

```
data GameTree = GameTree State [(Move, GameTree)]
```

Top-down solution

```
data GameTree = GameTree State [(Move, GameTree)]
data OutcomeTree a
  = Outcome a
```

Top-down solution

```
data GameTree = GameTree State [(Move, GameTree)]
data OutcomeTree a
  = Outcome a
  | OutcomeTree [(Move, OutcomeTree a)]
```

Top-down solution

```
data GameTree = GameTree State [(Move, GameTree)]
data OutcomeTree a
  = Outcome a
  | OutcomeTree [(Move, OutcomeTree a)]

findBestMove :: Int -> State -> Maybe Move
findBestMove depth initialState =
```

Top-down solution

```
data GameTree = GameTree State [(Move, GameTree)]
data OutcomeTree a
  = Outcome a
  | OutcomeTree [(Move, OutcomeTree a)]

findBestMove :: Int -> State -> Maybe Move
findBestMove depth initialState =
  _______ (______________________
    (______________ (______________ initialState)))
```

Top-down solution

```
data GameTree = GameTree State [(Move, GameTree)]
data OutcomeTree a
  = Outcome a
  | OutcomeTree [(Move, OutcomeTree a)]

findBestMove :: Int -> State -> Maybe Move
findBestMove depth initialState =
  _______ (______________________
    (_______________ (unfoldGameTree initialState)))
```

Top-down solution

```
data GameTree = GameTree State [(Move, GameTree)]
data OutcomeTree a
  = Outcome a
  | OutcomeTree [(Move, OutcomeTree a)]

findBestMove :: Int -> State -> Maybe Move
findBestMove depth initialState =
  _______ (_______________________
    (cutoffAt depth (unfoldGameTree initialState)))
```

Top-down solution

```
data GameTree state = GameTree state [(Move, GameTree)]
data OutcomeTree outcome
  = Outcome outcome
  | OutcomeTree [(Move, OutcomeTree outcome)]

findBestMove :: Int -> State -> Maybe Move
findBestMove depth initialState =
  _______ (toOutcomeTree estimate
    (cutoffAt depth (unfoldGameTree initialState)))
```

Top-down solution

```
data GameTree = GameTree State [(Move, GameTree)]
data OutcomeTree a
  = Outcome a
  | OutcomeTree [(Move, OutcomeTree a)]

findBestMove :: Int -> State -> Maybe Move
findBestMove depth initialState =
  minimax (toOutcomeTree estimate
    (cutoffAt depth (unfoldGameTree initialState)))
```

What are types?

What are types?

1. A hint to the compiler to know which machine codes to use?
2. A representation of how data is stored?
3. A set of possible values?
4. A set of possible operations/behaviours?
5. The meaning of data?

What can we do with ADTs?

What can we do with ADTs?

1. Reason about **equivalent types**
2. **Identify mismatches** between
   a. state space and used types
   b. possible inputs and handlers
3. Make illegal states **unrepresentable**

Algebraic Data Types...

...make illegal states unrepresentable.

Some use cases for Algebraic Data Types

1. State machines
2. Events
3. Commands
4. Abstract Syntax Trees
5. Composite data types

Product types are structs/records

```
data Student = Student Name Grade
data Grade = A | B | C | D
```

Product types are structs/records

```
data Student = Student { name :: Name, grade :: Grade }
data Grade = A | B | C | D
```

Product types are structs/records

```haskell
data Student = Student { name :: Name, grade :: Grade }
data Grade = A | B | C | D
```

```java
public final class Student {
  private final String name;
  private final Grade grade;
}
public enum Grade { A, B, C, D }
```

Sum types in Haskell

```haskell
data Result a
  = Success a
  | Failure String
```

Sum types in Haskell

```haskell
data Result a
  = Success a
  | Failure String

main :: IO ()
main = do
  let result = Success 42
  case result of
    Success value -> print value
    Failure message -> putStrLn ("Error: " ++ message)
```

Sum types in Java (using Visitor pattern)

```
public abstract class Result<A> {

    public abstract <R> R accept(Visitor<A,R> visitor);

    public interface Visitor<A,R> {
        R visit(Success<A> result);
        R visit(Failure<A> result);
    }

    public static class Success<A> extends Result<A> {
        public final A value;
        public Success(A value) { this.value = value; }
        @Override
        public <R> R accept(Visitor<A,R> visitor) { return visitor.visit(this); }
    }

    public static class Failure<A> extends Result<A> {
        public final String message;
        public Failure(String message) { this.message = message; }
        @Override
        public <R> R accept(Visitor<A,R> visitor) { return visitor.visit(this); }
    }
}
```

Sum types in Java (using Visitor pattern)

```java
public abstract class Result<A> {

    public abstract <R> R accept(Visitor<A,R> visitor);

    public interface Visitor<A,R> {
        R visit(Success<A> result);
        R visit(Failure<A> result);
    }

    public static class Success<A> extends Result<A> {
        public final A value;
        public Success(A value) { this.value = value; }
        @Override
        public <R> R accept(Visitor<A,R> visitor) { return visitor.visit(this); }
    }

    public static class Failure<A> extends Result<A> {
        public final String message;
        public Failure(String message) { this.message = message; }
        @Override
        public <R> R accept(Visitor<A,R> visitor) { return visitor.visit(this); }
    }
}
```

Sum types in Java (using Visitor pattern)

```
public abstract class Result<A> {

    public abstract <R> R accept(Visitor<A,R> visitor);

    public interface Visitor<A,R> {
        R visit(Success<A> result);
        R visit(Failure<A> result);
    }

    public static class Success<A> extends Result<A> {
        public final A value;
        public Success(A value) { this.value = value; }
        @Override
        public <R> R accept(Visitor<A,R> visitor) { return visitor.visit(this); }
    }

    public static class Failure<A> extends Result<A> {
        public final String message;
        public Failure(String message) { this.message = message; }
        @Override
        public <R> R accept(Visitor<A,R> visitor) { return visitor.visit(this); }
    }
}
```

Sum types in Java (using Visitor pattern)

```
public abstract class Result<A> {

  public abstract <R> R accept(Visitor<A,R> visitor);

  public interface Visitor<A,R> {
        R visit(Success<A> result);
        R visit(Failure<A> result);
  }

  public static class Success<A> extends Result<A> {
      public final A value;
      public Success(A value) { this.value = value; }
      @Override
      public <R> R accept(Visitor<A,R> visitor) { return visitor.visit(this); }
  }

  public static class Failure<A> extends Result<A> {
      public final String message;
      public Failure(String message) { this.message = message; }
      @Override
```

Sum types in Java (using Visitor pattern)

```
public abstract class Result<A> {

    public abstract <R> R accept(Visitor<A,R> visitor);

    public interface Visitor<A,R> {
        R visit(Success<A> result);
        R visit(Failure<A> result);
    }

    public static class Success<A> extends Result<A> {
        public final A value;
        public Success(A value) { this.value = value; }

        @Override

        public <R> R accept(Visitor<A,R> visitor) { return visitor.visit(this); }
    }

    public static class Failure<A> extends Result<A> {
        public final String message;
        public Failure(String message) { this.message = message; }
```

Pattern matching in Java via Visitor pattern

```
Result<Integer> result = new Result.Success(42);

String output = result.accept(new Result.Visitor<Integer, String>() {
    @Override
    public String visit(Result.Success<Integer> result) {
        return result.value.toString();
    }

    @Override
    public String visit(Result.Failure<Integer> result) {
        return result.message;
    }
});
```

Sum types and pattern matching in Scala

```
sealed abstract class Result[A];

case class Success[A](value : A) extends Result[A];
case class Failure[A](message : String) extends Result[A];
```

Sum types and pattern matching in Scala

```scala
sealed abstract class Result[A];

case class Success[A](value : A) extends Result[A];
case class Failure[A](message : String) extends Result[A];




val result : Result[Integer] = new Success(42)
result match {
  case Success(value)   => println(value.toString())
  case Failure(message) => println(message)
}
```

Sum types (variants) in C++

```
template <class A>
using Result = std::variant<Success<A>, Failure>;
```

Sum types (variants) in C++

```
template <class A> struct Success { A value; };

template <class A>
using Result = std::variant<Success<A>, Failure>;
```

Sum types (variants) in C++

```
template <class A> struct Success { A value; };
struct Failure { std::string message; };
template <class A>
using Result = std::variant<Success<A>, Failure>;
```

Sum types (variants) in C++

```
template <class A> struct Success { A value; };
struct Failure { std::string message; };
template <class A>
using Result = std::variant<Success<A>, Failure>;

int main() {
  Result<int> result = Success<int>{42};
  if (std::holds_alternative<Success<int>>(result)) {
    auto success = std::get<Success<int>>(result);
    std::cout << success.value << std::endl;
  } else {
    auto failure = std::get<Failure>(result);
    std::cout << failure.message << std::endl;
  }
  return 0;
}
```

Sum types and pattern matching in Rust

```
enum MyResult<A> {
    Success(A),
    Failure(String),
}
```

Sum types and pattern matching in Rust

```
enum MyResult<A> {
   Success(A),
   Failure(String),
}

fn main() {
    let result = MyResult::Success(42);
    match result {
        MyResult::Success(value)   => println!("value = {:?}", value),
        MyResult::Failure(message) => println!("Error: {:?}", message)
    }
}
```

Sum types in Swift

```
import Foundation

enum Result<A> {
  case Success(A)
  case Failure(String)
}
```

Sum types in Swift

```swift
import Foundation

enum Result<A> {
  case Success(A)
  case Failure(String)
}

let result = Result.Success(42)

switch result {
  case .Success(let value):
    print("value =", value);
  case .Failure(let message):
    print("Error:", message);
}
```

Homework (self-study)

1. Read **Chapters 1 and 2** of Learn Prolog Now!
   http://www.let.rug.nl/bos/lpn/lpnpage.php?pagetype=html&pageid=lpn-htmlch1
2. Work through the **exercises** from both chapters, using SWISH
   https://swish.swi-prolog.org
   (You may use SWISH's prototype version of Learn Prolog Now!:
   http://lpn.swi-prolog.org/lpnpage.php?pageid=online)

What was the most unclear part of the lecture for you?

See Moodle

References

1. CppCon 2016: Ben Deane “Using Types Effectively" https://youtu.be/ojZbFIQSdl8
2. std variant and the power of pattern matching - Nikolai Wuttke - Meeting C++ 2018 https://youtu.be/CELWr9roNno
3.
4. Java Pattern: Algebraic Data Types https://garciat.com/posts/java-adt
5.