15-150 Summer 2016
Homework 10

Out: Thursday, 16 June 2016
Due: Monday, 20 June 2016 at 23:59 EDT

1 Introduction

In this homework you will implement a game (TicTacToe) and you will write two game players (AlphaBeta and Jamboree)).

1.1 Getting The Homework Assignment

The starter files for the homework assignment have been distributed through our `git` repository, as usual.

1.2 Submitting The Homework Assignment

Submissions will be handled through Autolab, at `https://autolab.andrew.cmu.edu`.

In preparation for submission, your `hw/10` directory should contain a file named exactly `hw10.pdf` containing your written solutions to the homework.

To submit your solutions, run `make` from the `hw/10` directory (that contains a `code` folder and a file `hw10.pdf`). This should produce a file `hw10.tar`, containing the files that should be handed in for this homework assignment. Open the Autolab web site, find the page for this assignment, and submit your `hw10.tar` file via the "Handin your work" link.

This homework will be partially autograded. When you submit your code some *very* basic tests are run. These are the public tests, and you will immediately see your score on these. After the final submission deadline, we will run a more comprehensive suite of private tests on your code. Your final score will be a function of your public score, private score, and any manual grading we perform. Non-compiling code will automatically receive a 0.

Remember that your written solutions must be submitted in PDF format—we do not accept MS Word files or other formats.

All the code that you want to have graded for this assignment should be contained in `game/tictactoe.sml`, `game/alphabeta.sml`, `game/pokemon.sml`, and `game/jamboree.sml` (with `tournament/estimate.txt` for usage in the class tournament if desired), and must compile cleanly. If you have a function that happens to be named the same as one of the

required functions but does not have the required type, your code will not compile in our environment.

1.3 Due Date

This assignment is due on Monday, 20 June 2016 at 23:59 EDT. Remember that you may use a maximum of one late day per assignment, and that you are allowed a total of three late days for the semester. You may submit as many times as you would like until the due date.

1.4 Methodology

You must use the five step methodology discussed in class for writing functions, for every function we asked you write in this assignment, as well as any substantial helper functions for those functions. Recall the five step methodology:

1. In the first line of comments, specify the type of the function.
2. In the second line of comments, specify via a `REQUIRES` clause any assumptions about the arguments to be passed to the function.
3. In the third line of comments, specify via an `ENSURES` clause what the function computes (what it returns when applied to an argument that satisfies the assumptions in `REQUIRES`).
4. Implement the function.
5. Provide testcases, generally in the format
   `val <return value> = <function> <argument value>`.

For example, for the factorial function presented in lecture:

```
(*  fact : int -> int
 *  REQUIRES:  n >= 0
 *  ENSURES: fact(n) ==> n! *)
fun fact (0 : int) : int = 1
  | fact (n : int) : int = n * fact(n-1)

(* Tests: *)
val 1 = fact 0
val 6 = fact 3
val 720 = fact 6
```

Please test functions which are part of the signature for a structure or functor in a separate testing structure. Test functions which appear in a structure or functor but are not part of the signature directly underneath the function, according to the five-step method.

For example:

```
signature FOO =
sig
  val fact : int -> int
end

structure Foo : FOO =
struct
  (*  fact : int -> int
   *  REQUIRES:  n >= 0
   *  ENSURES: fact(n) ==> n! *)
  fun fact (0 : int) : int = 1
    | fact (n : int) : int = n * fact(n-1)
end

structure FooTests =
struct
  val 1 = Foo.fact 0
  val 6 = Foo.fact 3
  val 720 = Foo.fact 6
end
```

2 Records

2.1 Overview

A record is a keyed data structure built into SML that allows us to create unordered tuples with named fields. Often times using a normal tuple can be confusing when it has many members. It can be hard to keep track of which member represents which value. Records solve this problem by adding names to each of their members. Importantly, each field has a set type which the value at the field must have.

2.2 They're Still Functional

Although records may sound much like a dictionary in Python, or an object in Java, they have one very important difference. Unlike dictionaries and objects, records are immutable data structures. This means that once a record has been created, its values stay as they are forever. We must create a completely new record everytime we would like to change the value of a field. Since records are immutable, they are okay to use in a functional setting as we always know the type of each field, and the value of any field cannot change due to side effects.

2.3 Creating a Record and Accessing Fields

To create a record we use a special syntax we have not used before. The following will create a record with two fields, cat and dog, with the values "meow" and "woof" at each field respectively: `{cat = "meow", dog = "woof"}`. Suppose we have bound that record to the variable R. We can get the value at the dog field by doing the following: `#dog(R)`. Likewise, the value at the cat field can be retrieved with: `#cat(R)`.

2.4 Pattern Matching with Records

Like most other data structures, we can pattern match on Records. The syntax is the same as that to create a record, except in place of a value for each field you put a pattern: `case R of {cat = x, dog = y}` binds the value of the cat field to x and the value of the dog field to y. Therefore x has the value "meow" and y has the value "woof". Since records are unordered, it is important that we include the field names when creating or pattern matching a record.

3 Game 1: Tic-Tac-Toe

3.1 Introduction

The popular Tic-Tac-Toe game is back, but in $n \times n$ fashion! None of this simplistic 3×3 stuff (well, unless you want it); we're going to allow for arbitrary square boards this time!

3.2 Tic-Tac-Toe

You are all likely familiar with this game, but for those who aren't:

Tic-Tac-Toe is a game where the goal is to mark any row, column, or either of the two diagonals of a given $n \times n$ board with all O's or X's. For this game, Maxie is O and Minnie is X, and Maxie always goes first. Any player may place their entry (and only their entry) onto an empty space on the board during their turn. Gameplay proceeds with each player taking a turn until the win conditions are met or until the board is filled.

Normally, when the board is filled, the game ends in a Draw. However, to simplify things, we'll say that Minnie wins when the board is filled to avoid having to clutter the signatures in your code with the extra Draw case. This ends up giving Minnie a huge advantage at the end of the day, but Maxie was being a bully anyways by wanting to go first all the time (what a jerk!).

4 Tic-Tac-Toe: Implementation of Types

4.1 State

For our implementation, an entry can either be an O or an X. The board is made up of tiles, with each tile being an entry option (NONE means there is an empty spot on the board). We represent the state of the game as the current board and whose turn it is.

```
datatype entry = O | X
type tile = entry option
type board = tile seq seq
type state = board * player
```

4.2 Moves

A location on the board can be easily represented as a pair of integers `(r,c)`, where `r` is the row and `c` is the column. Note that these are 0-indexed! A move is a tuple containing the location to place an entry on as well as the entry itself.

```
type location = int * int
type move = entry * location
```

4.3 Initial Board Size

The initial board size information can be retrieved from the options handed to the functor `TicTacToe`. The options structure ascribes to a signature `TTTCONSTS`, which contains a value `board_size`, a positive integer that determines the number of rows and columns. Use this information to help you generate the start state and test for valid moves.

5 Tic-Tac-Toe: Tasks

Task 5.1 (4 pts). In `tictactoe.sml`, create the `start` state with the info in `Settings`.

Task 5.2 (7 pts). Implement the function `is_valid_move : state * move -> bool`.
This takes in a state and move, and returns true if the move can be made on the board given in state and false otherwise.

Task 5.3 (8 pts). Implement the function `make_move : state * move -> state`.
This, when given a state and move, will make the move, assuming the move is valid.

Task 5.4 (8 pts). Implement the function `moves : state -> move seq`.
This, when given a state, evaluates to a sequence of possible moves for the current player.

Task 5.5 (8 pts). Implement the function `status : state -> status`.
This, when given a state, will evaluate to `In_play` if the game can still be continued and `Over(outcome)` if the game is over with outcome `outcome`.

Task 5.6 (16 pts). Implement `estimate : state -> Est.est`.
`estimate` returns an estimated score for a given state.
Some quick rules about programming this function:

- `estimate` must **NOT** ever create nor examine future states. You may simulate moves at most one turn ahead as long as you never explicitly create a `state` (but you'll probably not need to do that for this game).
- `estimate` must **NOT** ever manipulate or hold stateful information. If you don't know what I'm referring to here, you're probably fine; this is for those people who might be thinking of using references, etc.

Failure to abide by these rules results in an automatic 0 for this task. If you are confused by what this means, please don't hesitate to ask on Piazza!

You will receive full credit as long as `estimate` performs better than an estimator that returns the maximum length of a chain of O's or X's for a particular player. Of course, a more refined evaluation algorithm will yield a more competent and challenging computer opponent! Your implementation of `estimate` should work on all valid game states, whether it is in play or it is a terminal state (leaf in the search tree). Remember that relatively lower scores should indicate Minnie is winning and relatively higher scores should indicate Maxie is winning. Your `estimate` should reflect this.

Please describe your implementation and the strategies you considered in a few sentences in a comment.

For testing, you can play the game in the REPL (see `lib/game/runtictactoe.sml`):

```
- CM.make "sources.cm";
- TicTacToe_HvH.go();
```

5.1 Introduction to Views

As we have discussed many times, lists operations have bad parallel complexity, but the corresponding sequence operations are much better. However, sometimes you want to write a sequential algorithm (e.g. because the inputs aren't very big, or because no good parallel algorithms are known for the problem). Given the sequence interface so far, it is difficult to decompose a sequence as "either empty, or a cons with a head and a tail." To implement this using the sequence operations we have provided, you have to write code that would lose style points:

```
case Seq.length s of
     0 =>
   | _ => ... (Seq.hd s) and (Seq.tl s) ...
```

We have solved this problem using a *view*. This means we put an appropriate datatype in the signature, along with functions converting sequences to and from this datatype. This allows us to pattern-match on an abstract type, while keeping the actual representation abstract. For this assignment, we have extended the `SEQUENCE` signature with the following members to enable viewing a sequence as a list:

```
datatype 'a lview = Nil | Cons of 'a * 'a seq
val showl : 'a seq -> 'a lview
val hidel : 'a lview -> 'a seq
(* invariant: showl (hidel v) ==> v *)
```

Because the datatype definition is in the signature, it can be used outside the abstraction boundary. The `showl` and `hidel` functions convert between sequences and list views. The following is an example of using this view to perform list-like pattern matching:

```
case Seq.showl s of
    Seq.Nil => ... (* Nil case *)
  | Seq.Cons (x, s') => ... uses x and s' ... (* Cons case *)
```

Note that the second argument to `Cons` is another `'a seq`, *not* an `lview`. Thus, `showl` lets you do one level of pattern matching at a time: you can write patterns like `Seq.Cons(x,xs)` but not `Seq.Cons(x,Seq.Nil)` (to match a sequence with exactly one element). We have also provided `hidel`, which converts a view back to a sequence—`Seq.hidel (Seq.Cons(x,xs))` is equivalent to `Seq.cons(x,xs)` and `Seq.hidel Seq.Nil` is equivalent to `Seq.empty()`.

`SEQUENCE` also provides a tree view:

```
datatype 'a tview = Empty | Leaf of 'a | Node of 'a seq * 'a seq

val showt : 'a seq -> 'a tview
val hidet : 'a tview -> 'a seq
```

that lets you pattern-match on a sequence as a tree. Such a pattern-match is essentially a `Seq.mapreduce`, but sometimes it is nice to write in pattern-matching style.

5.2 Look and Say, Revisited

In the next task, you will use list views to implement a slight variant of the look and say operation from homework 3, now on sequences. As specified on that homework, the `look_and_say` function transforms an `int list` into the `int list` that results from "saying" the numbers in the sequence such that runs of the same number are combined. We will generalize this by transforming an `'a Seq.seq` into an `(int * 'a) Seq.seq` with one pair for each run in the argument sequence. The `int` indicates the length of the run, and the value of type `'a` is the value repeated in the run. In order to test arguments for equality, `look_and_say` takes a function argument of type `'a * 'a -> bool`.

The following examples demonstrate the behavior of the function when given a function, `streq`, that tests strings for equality:

```
look_and_say streq <"hi","hi","hi"> ==> <(3,"hi")>
look_and_say streq <"bye","hi","hi"> ==> <(1,"bye"), (2, "hi")>
```

Task 5.7 (10 pts). Use the list view of sequences to write the function

```
look_and_say : ('a * 'a -> bool) -> 'a Seq.seq -> (int * 'a) Seq.seq
```

in the `LookAndSay` structure in `las.sml`.

We can also use sequences to model other abstract types; for instance a cycle, or cyclical list, is a sequence of elements with no "beginning" or "end". We can represent this with a sequence by remembering that the "last" element is adjacent to the "first" element.

Task 5.8 (10 pts). Using your `look_and_say`, write the function

```
las_wrap : ('a * 'a -> bool) -> 'a Seq.seq -> (int * 'a) Seq.seq
```

which returns the look-and-say for a sequence interpreted as a cycle. If the first and last elements of the sequence are part of a single run that wraps around, you should put this run at the *front* of the resulting sequence.

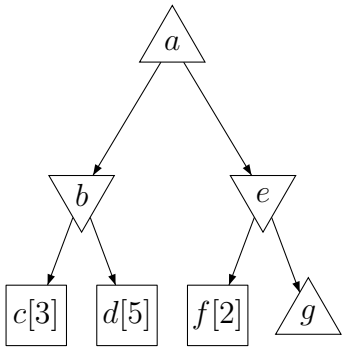
As an example, we have

```
las_wrap streq <''bye'',''hi'',''hi'',''bye''> ==> <(2,''bye''),(2,''hi'')>
las_wrap streq <''bye'',''hi'',''bye'',''hi''> ==>
                              <(1,''bye''),(1,''hi''),(1,''bye''),(1,''hi'')>
```

6 Alpha-Beta Pruning

In lecture, we implemented the MiniMax game tree search algorithm: each player chooses the move that, assuming optimal play of both players, gives that player the best possible score (highest for Maxie, lowest for Minnie). This results in a relatively simple recursive algorithm, which searches the entire game tree up to a fixed depth. However, it is possible to do better than this!

Consider the following game tree:



Maxie nodes are drawn as an upward-pointing triangle; Minnie nodes are drawn as a downward-pointing triangle. Each node is labeled with a letter, for reference below. The leaves, drawn as squares, are also labeled with their values (e.g., given by the estimator).

Let's search for the best move from left to right, starting from the root a. If Maxie takes the left move to b, then Maxie can achieve a value of 3 (because Minnie has only two choices, node c with value 3 and node d with value 5). If Maxie takes the right move to e, then Minnie can take the her left move to f, yielding a value of 2. But this is worse than what Maxie can already achieve by going left at the top! So Maxie already knows to go left to b rather than right to e. So, there is no reason to explore the tree g (which might be a big tree and take a while to explore) to find out what Minnie's value for e would actually be: we already know that the value, whatever it is, will be less than or equal to 2, and this is enough for Maxie not to take this path.

$\alpha\beta$-pruning is an improved search algorithm based on this observation. In $\alpha\beta$-pruning, the tree g is *pruned* (not explored). This lets an algorithm explore more of the relevant parts of the game tree in the same amount of time.

6.1 Setup

In $\alpha\beta$-pruning, we keep track of two bounds, representing the best (that is, highest for Maxie, and lowest for Minnie) score that can be guaranteed for each player based on the parts of the tree that have been explored so far. α is the highest guaranteed score for Maxie, and β is the lowest guaranteed score for Minnie. Any values in between α and β can be thought of as potential game values both players would like to have compared to their current best desirable score.

Note: We maintain the invariant that $\alpha < \beta$.

Consider a node visited in the search tree. This node represents a game state s. The search attempts to find the best possible move from s. Suppose the possible moves are $\{m_1, \ldots, m_k\}$. For each move m_i, the algorithm could make the move, resulting in a child node representing some game state s_i, then recursively compute a value v_i for that child. The MiniMax algorithm does compute each v_i. The value v of the node representing s is then simply $v = \max\{v_1, \ldots, v_k\}$ when at a Maxie node and $v = \min\{v_1, \ldots, v_k\}$ when at a Minnie node. The best possible move is whatever m_i gives rise to this value v.

In the $\alpha\beta$-pruning algorithm, there may be no need to actually make all moves m_i and compute all values v_i. To see this, consider a game state represented by a Maxie node in the search tree. Suppose the $\alpha\beta$ interval is (α, β). If ever $v_i \geq \beta$, then Maxie knows the search need never get to this game state. Why? Because Minnie could disallow it, since Minnie can achieve value β elsewhere in the tree.

Consequently, at a Maxie node, the $\alpha\beta$ algorithm recursively computes values $v_1, v_2, \ldots, v_i$ until it finds some i such that $v_i \geq \beta$, at which point it returns v_i to its calling function. In this case, the algorithm *does not* consider any remaining moves. If the condition $v_i \geq \beta$ never occurs, then the function returns $v = \max\{v_1, \ldots, v_k\}$ as with normal MiniMax.

Important: When making the recursive calls, Maxie updates the $\alpha\beta$ interval *for the recursive calls*. Specifically, when making move m_i and evaluating the node for state s_i, the $\alpha\beta$ interval is (α_i, β), using $\alpha_1 = \alpha$ and $\alpha_{j+1} = \max\{\alpha_j, v_j\}$ for $j \geq 1$.

This reasoning is symmetric for a Minnie node, now using the test $v_i \leq \alpha$ as a stopping criterion and updating a sequence of β_i values for the recursive calls.

The whole process starts off with the $\alpha\beta$ interval $(-\infty, +\infty)$.

Please refer to the slides from Lecture 22 for a visual example.

6.2 Tasks

We have provided starter code in `alphabeta.sml`. As in MiniMax, we need to return not just the value of a node, but the move that achieves that value, so that at the top we can select the best move.

This gives rise to the following type:

```
type edge  = G.move * G.est
```

Here `G.est` is a datatype that models players winning, as well as numerical estimates to indicate which player appears to have an advantage.

The function `search` computes the best edge, meaning a pair consisting of a move to a child along with the value of that child. (The function returns this best edge as an option.)

The function `evaluate` computes values of nodes, meaning a value of type `G.est`.

To evaluate each node, it is necessary to search some or all of the node's children and use information from those children to help return an appropriate value for the node itself.

An $\alpha\beta$ bound is defined as follows:

```
datatype bound = NEGINF | Bound of G.est | POSINF
type alphabeta = bound * bound
```

Note that we arbitrarily create `NEGINF` and `POSINF` to represent $-\infty$ and ∞, respectively.

It is also useful to define the following ordering:

```
datatype orderAB = BELOW | INTERIOR | ABOVE
fun compareAB ((a,b) : alphabeta) (v : G.est) : orderAB = ...
```

You'll want to use `compareAB` whenever you need to test whether an estimate is within the current $\alpha\beta$ bounds at a particular node.

Finally, we have provided four functions that you may find useful for debugging:

```
valueToString : value -> string
edgeToString : edge -> string
boundToString : bound -> string
abToString : alphabeta -> string
```

Task 6.1 (8 pts). Define the function

```
fun compareAB ((a,b) : alphabeta) (v : G.est) : orderAB = ...
```

that determines where `v` lies relative to the given $\alpha\beta$ bounds, as described in the specs for that function in the file `alphabeta.sml`.

Task 6.2 (18 pts). Define the function

```
fun search (d : int) (ab : alphabeta) (s : G.state) : edge option = ...
```

that uses the current $\alpha\beta$ bounds `ab` to find the best edge from the node for game state `s`, assuming search depth `d` > 0. (See again the discussion on page 11.) We have also given you the stubs for two helper functions that do the actual searching.

Task 6.3 (18 pts). Define the function

```
fun evaluate (d : int) (ab : alphabeta) (s : G.state) : G.est = ...
```

that determines the value of the current node corresponding to state `s` under non-negative search depth `d` and the $\alpha\beta$ bounds `ab`. Note that `search` and `evaluate` are mutually recursive! Also remember that when the depth is 0 or the game is over, `evaluate` should immediately estimate the current game state or return a value indicating a win, respectively.

You might find the views discussed in the previous section useful for these tasks.

Don't forget that the incoming $\alpha\beta$ interval affects the output of `search`, via proper pruning.

Task 6.4 (4 pts). Define the function

```
fun next_move (s : G.state) : move = ...
```

that returns the best move going out from `s`. Recall that `next_move` is a function specified in the `PLAYER` signature. In order to implement this function, you should search using $\alpha\beta$-pruning with the initial search depth as specified in `Settings` and with the initial $\alpha\beta$ range appropriately chosen.

Testing: You do *not* need to provide tests for your individual functions. You will however want to test your code yourself, by running some games.

Explicit Games for Testing: We have provided a functor `ExplicitGame` that makes a game from a given game tree, along with two explicit games, `HandoutBig` and `HandoutSmall` (see figures on page 14). These can be used for testing as follows:

```
- structure TestBig =
    AlphaBeta(struct structure G = HandoutBig val search_depth = 4 end);
- TestBig.next_move HandoutBig.start;
Estimating state e[3]
Estimating state f[5]
Estimating state h[2]
Estimating state l[10]
Estimating state m[4]
Estimating state q[2]
val it = 0 : HandoutBig.move

structure TestSmall =
    AlphaBeta(struct structure G = HandoutSmall val search_depth = 2 end);
- TestSmall.next_move HandoutSmall.start;
Estimating state c[3]
Estimating state d[6]
Estimating state e[~2]
Estimating state g[6]
Estimating state h[4]
Estimating state i[10]
Estimating state k[1]
val it = 1 : HandoutSmall.move
```

The search depths of `2` and `4` here are important, because an explicit game errors if it tries to estimate in the wrong place.

For these explicit games, `estimate` prints the states it visits, so you can see what terminal states (leaves) are visited, as indicated above. You may additionally wish to annotate your code so that it prints out traces.

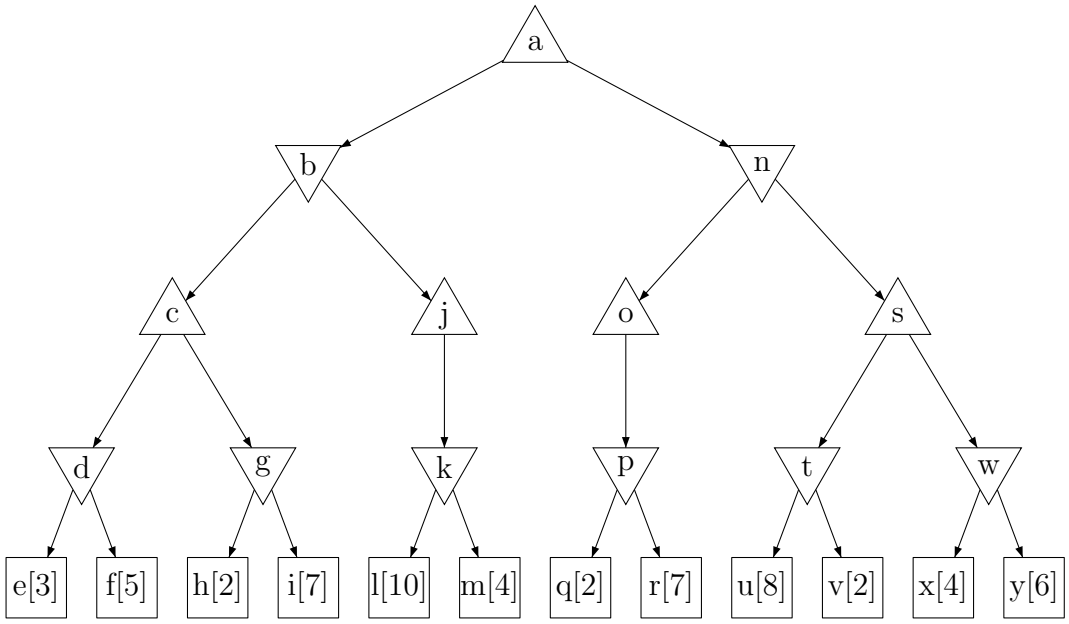


Figure 1: Tree for `HandoutBig`.

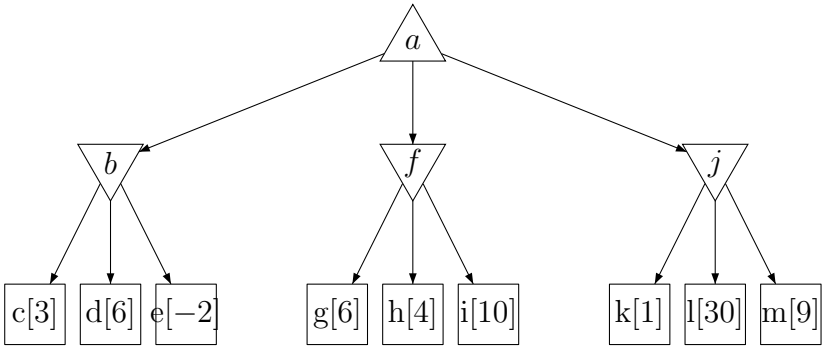


Figure 2: Tree for `HandoutSmall`.

7 Jamboree

$\alpha\beta$-pruning is entirely sequential, because you update $\alpha\beta$ as you search across the children of a node, which creates a dependency between children. On the other hand, MiniMax is entirely parallel: you can evaluate each child in parallel, because there are no dependencies between them. This is an example of a *work-span tradeoff*: MiniMax does more work, but has a better span, whereas $\alpha\beta$-pruning does less work, but has a worse span.

The *Jamboree*[1] algorithm manages this tradeoff by evaluating *some* of the children sequentially, using the $\alpha\beta$ algorithm, and then (if there was no pruning) the remainder in parallel, combining the results to find the optimal edge. Depending on the parallelism available in your execution environment, you can choose how many children to evaluate sequentially, in order to prioritize work or span.

In the file `jamboree.sml`, you will implement a functor

```
functor Jamboree (Settings : sig
                                structure G : GAME
                                val search_depth : int
                                val prune_percentage : real
                             end) : PLAYER
```

`prune_percentage` is assumed to be a real number between 0.0 and 1.0. For each[2] node, `100.0 * prune_percentage` percent of the children are evaluated sequentially using $\alpha\beta$, after which the remaining children are evaluated in parallel.
For example, with `prune_percentage` = 0.0, Jamboree performs just like MiniMax, and with `prune_percentage` = 1.0, Jamboree performs completely like sequential $\alpha\beta$-pruning.

7.1 Tasks

Task 7.1 Copy `compareAB` from your $\alpha\beta$-pruning implementation into `jamboree.sml`, plus whatever helper functions you might find useful.

Task 7.2 (5 pts). Define the function
`splitMoves : Game.state -> (Game.move seq * Game.move seq)`
which, when given game state `s`, splits the possible moves at `s` into two sequences, `(abmoves, mmmoves)`, as a function of the `prune_percentage`. In particular, if `s` has `n` moves, `abmoves` should contain the first `(Real.floor (prune_percentage * (real n)))` moves, and `mmmoves` should contain whatever is left over.

[1] A parallelized version of the original Scout $\alpha\beta$ algorithm in 1980, Jamboree is so named for running multiple "scouts" in parallel.

[2] This split occurs at every tree node `search`ed.

Task 7.3 (25 pts). Define the functions:

```
fun search (depth : int) (ab : alphabeta) (s : Game.state) : edge option = ...
and evaluate (depth : int) (ab : alphabeta) (s : Game.state) : G.est = ...
```

using the Jamboree algorithm.

The specs for `search` and `evaluate` are as they are for $\alpha\beta$-pruning and MiniMax, now combined with the percentage tradeoff as described above. The function `search` should divide up the `moves` from `s` into `abmoves` and `mmmoves` based on the specified percentage, then call the functions `maxisearch` and `minisearch` (caution: the types of these functions in `jamboree.sml` have changed from those in `alphabeta.sml`).

The algorithm should process `abmoves` sequentially, updating $\alpha\beta$ as in an $\alpha\beta$-pruning implementation. When there are no more `abmoves` (and if there has been no pruning), then the algorithm should evaluate `mmmoves` in parallel, and combine the results from these two methods of search (by either maximizing or minimizing values, as appropriate for the given player).

(Again, you may find some of the functions in the `SEQUENCE` signature that you haven't used before helpful.)

Task 7.4 (1 pts). Define `next_move`.

Testing: Test your code as with the "alphabeta" part of the assignment.