15-150 Spring 2012
Homework 09

Out: 10 April, 2012
Due: 25 April, 2012, 09:00am

1 Introduction

In this homework you will write a game and a game player using the techniques you have been learning in class. There are two major parts: writing the representation of the game, and writing a parallel version of the popular alpha-beta pruning algorithm. This will also continue to test your ability to use and understand the module system.

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

To submit your solutions, place your modified `las.sml`, `connect4.sml`, `runconnect4.sml`, `alphabeta.sml`, and `jamboree.sml` files in your handin directory on AFS:

```
/afs/andrew.cmu.edu/course/15/150/handin/<yourandrewid>/hw09/
```

Your files must be named exactly: `las.sml`, `connect4.sml`, `runconnect4.sml`, `alphabeta.sml`, and `jamboree.sml`. After you place your files in this directory, run the check script located at

```
/afs/andrew.cmu.edu/course/15/150/bin/check/09/check.pl
```

then fix any and all errors it reports.

Remember that the check script is *not* a grading script—a timely submission that passes the check script will be graded, but will not necessarily receive full credit.

Your `las.sml`, `connect4.sml`, `runconnect4.sml`, `alphabeta.sml`, and `jamboree.sml` files must contain all the code that you want to have graded for this assignment and 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, it will not be graded. Modules must ascribe to the specified signatures or they will not be graded.

1.3 Methodology

You must include types and purposes for every function you write on this assignment. However, this is a larger project, and can be tested in different ways: in addition to writing tests for each function, you can play your game, and test your game tree search on the small example games we have provided. Thus, we will grade you only on the correctness of your code, and not allocate any points directly to testing. Of course, it is still essential that you test your code: in a project this big, you will have bugs!

1.4 Style

We will continue grading this assignment based on the style of your submission as well as its correctness. Please consult course staff, your previously graded homeworks, or the published style guide as questions about style arise.

1.5 Due Date

This assignment is due on 25 April, 2012, 09:00am. Remember that this deadline is final and that we do not accept late submissions.

1.6 The SML/NJ Build System

The support code for this assignment includes a substantial amount of starter code distributed over several files. The compilation of this is again orchestrated by CM through the file `sources.cm`. Instructions on how to use CM can be found in the previous homework handout.

2 Views

For this problem, you will want to use the *list view* for sequences. The `SEQUENCE` signature contains the following components, which we have not talked about until now:

```
signature SEQUENCE =
sig
  ...

  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 *)
end
```

See the Lecture 22-23 notes for an introduction to views.

You will use list views to implement a slight variant of the look and say operation from homework 3 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 2.1 (10%). 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`.

3 Connect 4

3.1 Description

Connect 4 is a strategy game played on a grid. Two players—Maxie (X) and Minnie (O)—take turns dropping a piece into a column. The first player to have four markers in a row, either horizontally, vertically, or diagonally wins. The choice of board size is somewhat arbitrary, so your implementation will be parameterized over the board size, but will always use four pieces in a row as the winning condition.

For example, Figure 1 shows a few example plays, starting from a board where Maxie has a piece in column 2 and a piece in column 3, and Minnie has a piece in column 3 (on top of Maxie's) and a piece in column 4. Next, Maxie decides to make the move `0` (drop a piece in column 0), which falls to the bottom of column 0. Maxie now has three in a row, and can win by playing in column 1 if it is still free on Maxie's next turn. But Minnie decides to block, by playing in column 1.

3.2 Implementation

The support code for this homework contains the `GAME` code discussed in class (see the lecture 22/23 notes for more details). Part of an implementation of the `GAME` signature for Connect 4 is provided in `games/connect4.sml`. It uses the following representation of game states:

```
datatype position = Filled of player | Empty

(* (0,0) is bottom-left corner *)
datatype c4state = S of (position Matrix.matrix) * player
type state = c4state

type move = int (* cols are numbered 0 ... numcols-1 *)
```

A board is represented by a matrix. See the signature `MATRIX` (in `matrix/matrix.sig`) for matrix operations—for example, the `rows`, `cols`, `diags1`, and `diags2` functions may be helpful. In particular, the bottom-left corner of the board thought of as the origin, so matrix indices `(x,y)` can be thought of as x-y coordinates in the plane. E.g. in the following board:

```
 0 1 2 3 4 5 6
---------------
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | |O| | | |
| | |X|X|O| | |
 0 1 2 3 4 5 6
---------------
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | |O| | | |
| | |X|X|O| | |


Maxie, please type your move: 0
Maxie decides to make the move 0 in 3 seconds.

 0 1 2 3 4 5 6
---------------
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | |O| | | |
|X| |X|X|O| | |


Minnie decides to make the move 1 in 0 seconds.

 0 1 2 3 4 5 6
---------------
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | |O| | | |
|X|O|X|X|O| | |
```

Figure 1: A few plays of Connect 4

Minnie (O) has pieces in spaces $(4,0)$ and $(3,1)$. In the matrix, each position is either `Filled` with a `player`'s piece, or `Empty`.

Task 3.1 (50%). Complete the implementation of the `Connect4` functor in `connect4.sml`. This takes a structure ascribing to the `CONNECT4_BOARD_SPEC` signature specifying the dimensions of the board.

Some advice:

- We have implemented most of the parsing and printing for you, but the move parser relies on a function

  ```
  lowestFreeRow :  state -> int -> int option
  ```

  where `lowestFreeRow s i` returns SOME(the index of the lowest free row in column `i`)—i.e. the row a piece would fall into—or `NONE` if the column is full. You will want to use this function elsewhere as well.

- We have provided some additional sequence operations that are helpful for this problem in the structure `SeqUtils`, in `sequtils.sml`.

- You may find `look_and_say` helpful for your implementation of `status`.

- `estimate` is pretty open-ended. A very naïve estimator is to award some number of points for each run of markers in a row; for instance, three in a row might be worth 64 points, two in a row 16, and so on. Runs for Maxie are awarded positive points, whereas runs for Minnie are awarded negative points. **If your estimator is at least this smart, you will receive full credit.**

  A more sophisticated estimator might take the following things into account:

  - whether or not a run can possibly lead to a win. E.g.

    ```
    O X X X _
    ```

    is better than

    ```
    O X X X O
    ```

  - how many ways a run can lead to a win. E.g.

    ```
    _ X X X _
    ```

    is even better!

  - non-runs can be as good as runs. E.g. the following is also “three in a row”:

```
        X _ X X
```

   – how many moves it will take to complete a run, based on its height off the ground.
   – the importance of blocking the other player's moves

However, keep in mind that writing a clever estimator is for fun, and not required for credit on this assignment. In particular, it's more important for you to work on $\alpha\beta$-pruning and Jamboree than to spend time improving your estimator.

3.3 Running Your Game

The file `runconnect4.sml` uses the `Referee` to construct a Connect 4 match, with Maxie as a human player and Minnie as Minimax. Thus, you can run your game with:

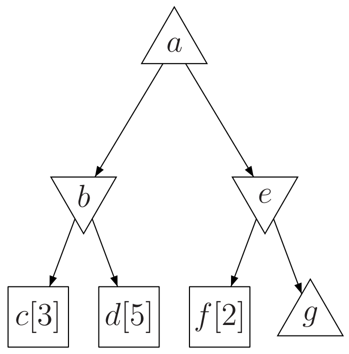
```
- CM.make "sources.cm";
- C4_HvMM.go();
```

You can write other referee applications to test different variations on the game: different board sizes, different search depths, different game tree search algorithms (such as $\alpha\beta$-pruning and Jamboree below).

4 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 them 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 labelled with a letter, for reference below. The leaves, drawn as squares, are Maxie nodes and are also labelled 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 you explore more of the relevant parts of the game tree in the same amount of time.

4.1 Setup

In $\alpha\beta$-pruning, we keep track of two values, 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.

For each node, $\alpha\beta$-pruning computes a *result*, which is either a number (in particular, `Game.estimate`), or one of two special messages, `Pruned` and `ParentPrune`:

- A result `ParentPrune` means "the parent of this node is not helpful to the grandparent of this node". In the above example, the result of f is `ParentPrune`, because the value 2 that the parent (Minnie) node can achieve is worse for the grandparent (Maxie) node than what Maxie can already achieve by going left from a.
- A result `Pruned` means "this node should be ignored by the parent." In the above example, the result of e is `Pruned`, signalling that e is not helpful to a.

$\alpha\beta$-pruning labels each node in the game tree with a result, according to the following spec:

> **Spec for $\alpha\beta$-pruning:** Fix a search depth and estimation function, so that both minimax and $\alpha\beta$-pruning are exploring a tree with the same leaves. Fix bounds α and β such that such that $\alpha < \beta$. Let MM be the minimax value of a node s. Then the $\alpha\beta$-pruning result for that node, AB, satisfies the following:
>
> - If $\alpha < MM < \beta$ then $AB = MM$.
> - If $MM \leq \alpha$ then
>   - $AB =$ `ParentPrune` if s is a Maxie node
>   - $AB =$ `Pruned` if s is a Minnie node
> - If $MM \geq \beta$ then
>   - $AB =$ `Pruned` if s is a Maxie node
>   - $AB =$ `ParentPrune` if s is a Minnie node

Roughly, α is what we know Maxie can achieve, and β is what we know Minnie can achieve. If the true minimax value of a node is between these bounds, then $\alpha\beta$-pruning computes that value. If it is outside these bounds, then $\alpha\beta$-pruning signals this in one of two ways, depending on which side the actual value falls on, and whose turn it is. Suppose that it's Maxie's turn.

- If the actual minimax value is less than α, the node should be labeled `ParentPrune`, which is an instruction to immediately label the *parent* Minnie node as `Pruned`, so that the enclosing Maxie grandparent ignores it. The reason: this Maxie node's value is worse for Maxie than what we already know Maxie can achieve, so it gives the enclosing Minnie node an option that Maxie doesn't want it to have. So the Maxie grand-parent should ignore this branch, independently of what the other siblings are. Node f in the above tree is an example.
- If the actual minimax value is greater than β, the $\alpha\beta$-pruning value should be `Pruned`, because the node is "too good". This is because the value is better, for Maxie, than what we already know Minnie can achieve, so Minnie won't make choices that bolead to this branch of the tree. Node d in the above tree is an example.

The labels for Minnie are dual.

Sometimes, no bounds on α and β are known (e.g., at the initial call, when you have not yet explored any of the tree). To account for this, we let α range over things that are either a number or are `Pruned`, where `Pruned` signals "no information". `Pruned` is treated as the *smallest* α, so $max(\texttt{Pruned}, \alpha) = \alpha$ for any α (`Pruned` means "don't use this node", so anything is better than it) and `Pruned` $\leq \alpha$ (`Pruned` is no bound at all, so anything is better than it). Dually, for β, we want `Pruned` to be the *biggest* β: `Pruned` $\geq \beta$ and $min(\beta, \texttt{Pruned}) = \beta$.

4.2 The Algorithm

The above spec doesn't entirely determine the algorithm (a trivial algorithm would be to run minimax and then adjust the label at the end). The extra ingredient is how the values of α and β are adjusted as the algorithm explores the tree. The key idea here is *horizontal propagation*.

Consider the case where we want to compute the value of a *parent* node (so it is not a leaf), given search bounds α and β based on the portion of the tree seen so far.

- To calculate the result for the parent node, you scan across the children of the node, recursively calculating the result of each child from left to right, using the $\alpha\beta$ for the parent as the initial $\alpha\beta$ for the first child. After considering each child:
  - If the result of the child is `ParentPrune`, you stop and label the parent with `Prune`, without considering the remaining children.
  - Otherwise, you use the child's value v to update the value of α *or* the value of β used as the bound for the next child: If the parent is a Maxie node, you update $\alpha_{new} = max(\alpha_{old}, v)$ (because α is what we know Maxie can achieve, which can be improved by v). Dually, if the parent is a Minnie node, you update $\beta_{new} = min(\beta_{old}, v)$ (because β is what we know Minnie can achieve, which can be improved by v). Then you continue processing the remaining children using the updated bounds.
- Once all children have been processed, the parent node is labeled using the final updated α (for Maxie) or β (for Minnie) as a *candidate value*, which is treated as the value MM in the . **Spec for $\alpha\beta$-pruning**: If the candidate value is within the bounds $\alpha\beta$ supplied for the parent node, then the node is labeled with the candidate value; otherwise, it is labeled with `Pruned`/`ParentPrune`.

  Note that, except via the returned value of a node, the updates to α and β made in children do not affect the α and β for the parent node—they are based on different information.

Leaves are labeled using the estimated/actual value as a candidate value, but again you must do a bounds check and label with `ParentPrune` or `Pruned` if the **Spec for $\alpha\beta$-pruning** requires it.

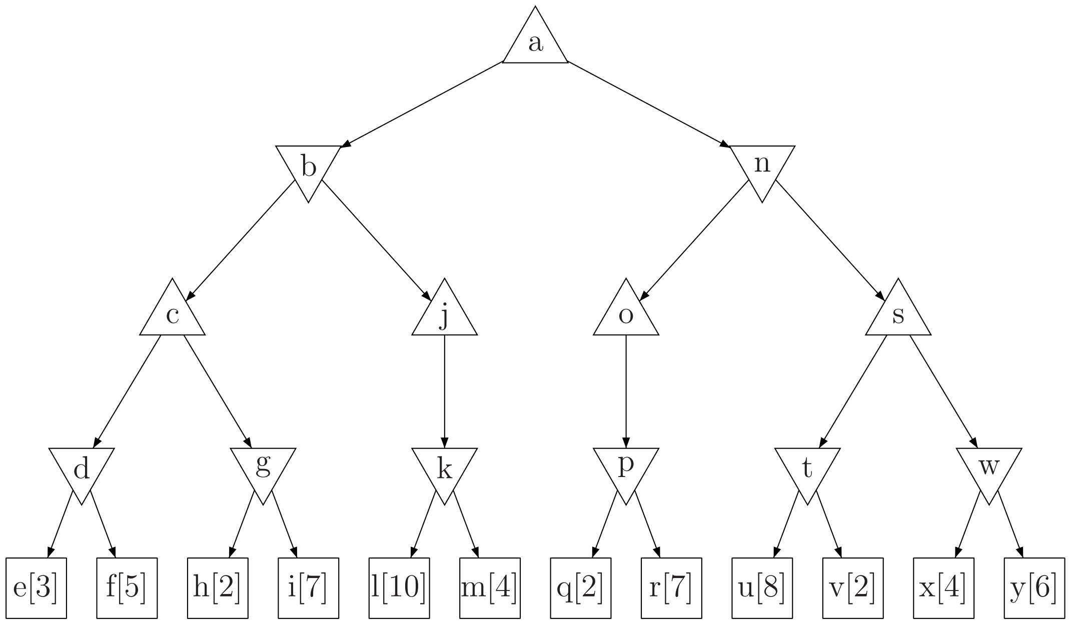


Figure 2: Extended $\alpha\beta$-pruning example

4.3 Extended Example

Figure 3 shows a trace of this algorithm running on the tree in Figure 2. As explained above, we use `Pruned` in the initial call to represent "no bound for α/β." The $c - d - g$ subtree is isomorphic to the above example tree, and shows how the algorithm runs on it. We start evaluating c with no bounds, so we recursively look at d, and then e, still with no bounds. The estimate for e is 3, which is trivially within the bounds, so the result of e is 3. Because d is a Minnie node, we update β to be $min(\texttt{Pruned}, 3)$, which is defined to be 3, for a recursive call on f. Since the estimate of f is 5, which is greater than the given β (which is 3), and it is a Maxie node, the result is `Pruned`, to signal that the enclosing Minnie node doesn't want this branch. Since the min of 3 and `Pruned` is still 3, the value of d is 3. Since c is a Maxie node, we update α to be $max(\texttt{Pruned}, 3)$, which is also 3. These bounds are passed down to g and then h. Since the actual value of h is 2, which is less than the given α, and it is a Maxie node, the value of h is `ParentPrune`: we know Maxie can get 3 in the left tree, and this branch alone gives Minnie the ability to get 2 here, so Maxie doesn't want to take it. Thus, the value of g is `Pruned`, *without even looking at* i. Then we update α to be $min(3, \texttt{Pruned})$, which is still 3. Since there are no other children, and since 3 is in the original bounds for c, it is the value of c.

```
Searching  state (Maxie,a) with ab=(Pruned,Pruned)

(* begin c-d-g sub tree *)
Evaluating state (Maxie,c) with ab=(Pruned,Pruned)
Evaluating state (Minnie,d) with ab=(Pruned,Pruned)
Evaluating state (Maxie,e) with ab=(Pruned,Pruned)
Result of (Maxie,e) is Guess:3
Evaluating state (Maxie,f) with ab=(Pruned,Guess:3)
Result of (Maxie,f) is Pruned
Result of (Minnie,d) is Guess:3
Evaluating state (Minnie,g) with ab=(Guess:3,Pruned)
Evaluating state (Maxie,h) with ab=(Guess:3,Pruned)
Result of (Maxie,h) is ParentPrune
Result of (Minnie,g) is Pruned
Result of (Maxie,c) is Guess:3
(* end c-d-g sub tree *)
Evaluating state (Maxie,j) with ab=(Pruned,Guess:3)
Evaluating state (Minnie,k) with ab=(Pruned,Guess:3)
Evaluating state (Maxie,l) with ab=(Pruned,Guess:3)
Result of (Maxie,l) is Pruned
Evaluating state (Maxie,m) with ab=(Pruned,Guess:3)
Result of (Maxie,m) is Pruned
Result of (Minnie,k) is ParentPrune
Result of (Maxie,j) is Pruned
Result of (Minnie,b) is Guess:3
Evaluating state (Minnie,n) with ab=(Guess:3,Pruned)
Evaluating state (Maxie,o) with ab=(Guess:3,Pruned)
Evaluating state (Minnie,p) with ab=(Guess:3,Pruned)
Evaluating state (Maxie,q) with ab=(Guess:3,Pruned)
Result of (Maxie,q) is ParentPrune
Result of (Minnie,p) is Pruned
Result of (Maxie,o) is ParentPrune
Result of (Minnie,n) is Pruned

Therefore value of (Maxie,a) is Guess:3, and best move is 0.

Terminals visited: 6
```

Figure 3: AB-Pruning trace for the above tree

4.4 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:

```
  type edge = (Game.move * Game.est)
```

From the above discussion, you might expect

```
  datatype result =
      BestEdge of edge
    | Pruned
    | ParentPrune
```

for representing the various possible results of evaluating a node. However, it will be useful to stage this into two datatypes:

```
  datatype value =
      BestEdge of edge
    | Pruned

  datatype result =
      Value of value
    | ParentPrune
```

This way, we can represent α and β by a `value`, with `Pruned` representing "no information". Because α and β are not `results`, the type system will force you to check for `ParentPrune` in the appropriate place.

The reason we use the same value `Pruned` to mean both "this subtree was pruned" and "no bound" (e.g. in the initial call) is that both meanings are consistent with the same ordering on `values`.

For α, we order the type `value` with `Pruned` at the bottom, and `BestEdge`'s ordered by the estimates in them. This order gives that $max(\texttt{Pruned}, \alpha) = \alpha$ for any α (`Pruned` means "don't use this node", so anything is better than it) and $\texttt{Pruned} \leq \alpha$ (`Pruned` is no bound at all, so anything is better than it). Dually, for β, we want $\texttt{Pruned} \geq \beta$ and $min(\beta, \texttt{Pruned}) = \beta$, which means we think of `Pruned` as the *top* of `value` for β.

We have provided four functions:

```
alpha_is_less_than (alpha : value, v : Game.est) : bool
maxalpha : value * value -> value

beta_is_greater_than (v : Game.est, beta : value) : bool
minbeta : value * value -> value
```

that implement these orderings.

Also we abbreviate

```
type alphabeta = value * value (* invariant: alpha < beta *)
```

Task 4.1 (2%). Define the function

```
fun updateAB (s : Game.state) (ab : alphabeta) (v : value) : alphabeta = ...
```

that updates the appropriate one of `ab` with the new value `v`, which should be thought of as the value of one of the children of `state`.

Task 4.2 (2%). Define the function

```
fun value_for (state : Game.state) (ab : alphabeta) : value = ...
```

that returns the appropriate one of `ab` for the player whose turn it is in `state`.

Task 4.3 (4%). Define the function

```
fun check_bounds ((alpha,beta) : alphabeta) (state : Game.state)
                 (incomingMove : Game.move) (e : Game.est) : result = ...
```

that takes a candidate value `e` (think of this as the MiniMax result MM) and returns the appropriate `result` according to the above **Spec for $\alpha\beta$-pruning**. The `incomingMove` is given because, when the value is in bounds, the result must be an `edge`, not just a number.

Task 4.4 (20%). Define the functions:

```
fun evaluate (depth : int) (ab : alphabeta) (s : Game.state)
             (incomingMove : Game.move) : result  = ...
and search (depth : int)
           (ab : alphabeta)
           (s : Game.state)
           (moves : Game.move Seq.seq)= ...
```

`search` may assume the depth is non-zero, the state is `In_play`, and that `moves` is a sequence of valid moves for `s`. `search` is responsible for evaluating the children of `s` given by the moves in `moves` and returning the appropriate value for the parent `s`.[1] **Hint: don't forget that the incoming α/β must be included as a possibility for the output of `search`, to get proper pruning. If your code is additionally visiting nodes u, v, x, y in the tree in Figure 2, check for this bug.**

[1]Note: We have not specified the result type for `search`: our solution uses `value`, but you are free to make `search` return a `result` if this makes more sense to you.

`evaluate` checks the boundary conditions (depth is 0, game is over) and, if not, searches. In any case, `evaluate` must ensure that the `result` it computes for `s` satisfies the above **Spec for $\alpha\beta$-pruning**.

Task 4.5 (2%). Define `next_move`.

Task 4.6 (2%). In `runconnect4.sml`, use the `Referee` to define a structure `C4_HvAB` that allows you to play Connect 4 against your $\alpha\beta$-pruning implementation.

Testing We have provided a functor `ExplicitGame` that makes a game from a given game tree, along with two explicit games, `HandoutSmall` (Figure 4) and `HandoutBig` (Figure 3). 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);
structure TestSmall : PLAYER?
- 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 terminals are visited, as indicated above. You may additionally wish to annotate your code so that it prints out traces, as above.

5 Jamboree

$\alpha\beta$-pruning is entirely sequential, because you update α/β 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.

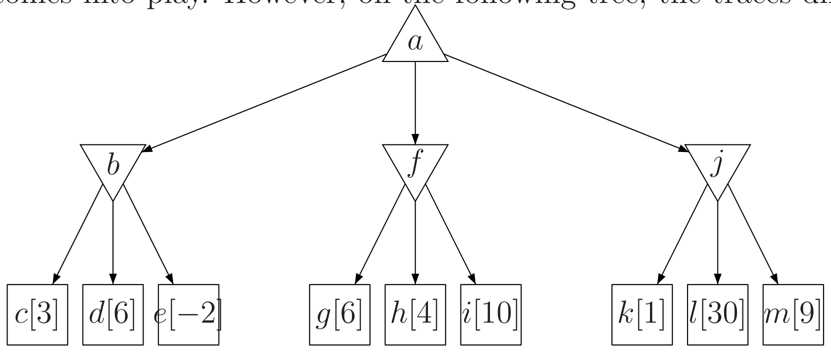
The *Jamboree* algorithm manages this tradeoff by evaluating *some* of the children sequentially, updating α/β, and then the remainder in parallel, using the updated information. Depending on the parallelism available in your execution environment, you can choose how many children to do sequentially 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 number between 0 and 1. For each node, `prune_percentage` percent of the children are evaluated sequentially, updating $\alpha\beta$, and the remaining children are evaluated in parallel. For example, with `prune_percentage` $= 0$, Jamboree specializes to MiniMax, and with `prune_percentage` $= 1$, Jamboree specializes to completely sequential $\alpha\beta$-pruning.

Suppose `prune_percentage` is `0.5`. For the tree in Figure 2, Jamboree will explore the tree in the same way as in Figure 3: no node has more than 2 children, so the restriction on pruning never comes into play. However, on the following tree, the traces differ:



See Figure 4 for the traces.

5.1 Tasks

Task 5.1 Copy `updateAB`, `value_for`, and `check_bounds` from your $\alpha\beta$-pruning implementation, as these will be unchanged.

Task 5.2 (10%). Define the functions:

```
fun evaluate (depth : int) (ab : alphabeta) (s : Game.state)
             (incomingMove : Game.move) : result  = ...
and search (depth : int) (ab : alphabeta) (s : Game.state)
           (abmoves : Game.move Seq.seq) (mmmoves : Game.move Seq.seq)
           = ...
```

using the Jamboree algorithm.

The spec for `evaluate` is as above, except that when it calls `search`, it should divide up the `moves` from `s` into `abmoves` and `mmmoves`. In particular, if `s` has `n` moves out of it, `abmoves` should contain the first `(floor (prune_percentage * n))` moves, and `mmmoves` whatever is left over.

`search` should process `abmoves` sequentially, updating α/β as you go, as in your $\alpha\beta$-pruning implementation. When there are no more `abmoves`, `search` should process `mmmoves` in parallel, and combine the max/min of their values with the incoming α/β to produce the appropriate value for `s`.

Task 5.3 (2%). Define `next_move`.

Jamboree with `prune_percentage` $= 0.5$, so $\alpha\beta$ are updated only after the first child (we round down):

```
Evaluating state (Minnie,b) with ab=(Pruned,Pruned)
Evaluating state (Maxie,c) with ab=(Pruned,Pruned)
Result of (Maxie,c) is Guess:3
Evaluating state (Maxie,d) with ab=(Pruned,Guess:3)
Result of (Maxie,d) is Pruned
Evaluating state (Maxie,e) with ab=(Pruned,Guess:3)
Result of (Maxie,e) is Guess:~2
Result of (Minnie,b) is Guess:~2
Evaluating state (Minnie,f) with ab=(Guess:~2,Pruned)
Evaluating state (Maxie,g) with ab=(Guess:~2,Pruned)
Result of (Maxie,g) is Guess:6
Evaluating state (Maxie,h) with ab=(Guess:~2,Guess:6)
Result of (Maxie,h) is Guess:4
Evaluating state (Maxie,i) with ab=(Guess:~2,Guess:6)
Result of (Maxie,i) is Pruned
Result of (Minnie,f) is Guess:4
Evaluating state (Minnie,j) with ab=(Guess:~2,Pruned)
Evaluating state (Maxie,k) with ab=(Guess:~2,Pruned)
Result of (Maxie,k) is Guess:1
Evaluating state (Maxie,l) with ab=(Guess:~2,Guess:1)
Result of (Maxie,l) is Pruned
Evaluating state (Maxie,m) with ab=(Guess:~2,Guess:1)
Result of (Maxie,m) is Pruned
Result of (Minnie,j) is Guess:1
Terminals visited: 9
Overall choice: move 1[middle]
```

$\alpha\beta$-pruning:

```
Evaluating state (Minnie,b) with ab=(Pruned,Pruned)
Evaluating state (Maxie,c) with ab=(Pruned,Pruned)
Result of (Maxie,c) is Guess:3
Evaluating state (Maxie,d) with ab=(Pruned,Guess:3)
Result of (Maxie,d) is Pruned
Evaluating state (Maxie,e) with ab=(Pruned,Guess:3)
Result of (Maxie,e) is Guess:~2
Result of (Minnie,b) is Guess:~2
Evaluating state (Minnie,f) with ab=(Guess:~2,Pruned)
Evaluating state (Maxie,g) with ab=(Guess:~2,Pruned)
Result of (Maxie,g) is Guess:6
Evaluating state (Maxie,h) with ab=(Guess:~2,Guess:6)
Result of (Maxie,h) is Guess:4
Evaluating state (Maxie,i) with ab=(Guess:~2,Guess:4)
Result of (Maxie,i) is Pruned
Result of (Minnie,f) is Guess:4
Evaluating state (Minnie,j) with ab=(Guess:4,Pruned)
Evaluating state (Maxie,k) with ab=(Guess:4,Pruned)
Result of (Maxie,k) is ParentPrune
Result of (Minnie,j) is Pruned
Terminals visited: 7
Overall choice: move 1[middle]
```

Figure 4: Traces for Tree 2

6 Extra Credit

6.1 PopOut

This extra credit task is in addition to, not in place of, the above. You will not get credit for the above if you only hand in in the extra credit versions. Extra credit will be taken into account when determining your final grade. This task will be worth somewhere in the vacinity of 25 Homework points.

PopOut is a variation on Connect 4, with one additional move: a player can pop out one of their own pieces from the bottom row, which shifts everything else in the column down. E.g. in the state (note: not a board that will arise during play)

```
 0 1 2 3 4 5 6
---------------
| | | | | | | |
| | | | | | | |
| | | | | | | |
|X| | | | | | |
|O|X|X|X| | | |
|X|O|X|X|O| | |
```

Maxie can pop out column 0 and win.

Task 6.1 In a file `popout.sml`, implement a functor

```
functor PopOut (Dim : CONNECT4_BOARD_SPEC) : GAME =
```

Be sure to include a comment at the top of the file describing the following:

1. What the input format is for a human player to enter a pop move.
2. How your estimator accounts for pops.