15-150 Fall 2011
Homework 10

Out: Thursday, Nov 17
Due: Tuesday, Nov 29 at 11:59 PM

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 Submission

This assignment comes with several SML files you should read but not modify. These files can all be extracted from the `hw10.tar.bz2` file. Much of the information you need about the types and functions in these SML files is summarized throughout this handout. To write your code solutions, you should only edit and submit the files `las.sml`, `connect4.sml`, and `jamboree.sml`.

To submit your solutions place these files in your handin directory, located at

`/afs/andrew.cmu.edu/course/15/150/handin/<yourandrewid>/assn10/`

Your source files must be named as above.

Your files must contain all the code that you want to have graded for this assignment 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, it will not be graded.

1.2 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, **there will be no points allocated testing (white-box tests on functions, or otherwise)**. This is a double-edged sword: there will

correspondingly be more points allocated to correctness. We leave it up to you to decide when testing individual functions is worth your while, and when it is better to do something more interactive. Because the interactive tests are extra-lingual (you look at the output on the screen), we won't ask you to hand them in for credit.

1.3 Naming Modules

In this assignment, we will ask you to write modules from scratch. Your modules must be named exactly as required in the handout. Correct code inside an incorrectly named structure *will not receive any credit*.

You also may not modify any of the signatures that we give you. We will test your code against the signatures that we handed out, so if you modify the signatures your code will not compile and you will not receive credit.

1.4 The SML/NJ Build System

The tarball 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

2.1 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 opertions 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 solve 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 the signature, they 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.

2.2 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 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`.

2.3 Reasoning Using Views

Views also give an induction principle for sequences, enabling us to reason about sequences using induction, as if they were lists or tree. For example, for the list view:

> Consider a predicate P(`s:'a Seq.seq`) on sequences.
> To prove $\forall$`s:'a Seq.seq`.$P(s)$, it suffices to show
>
> - P(`Seq.hidel Seq.Nil`)
> - For all `x : 'a` and `xs : 'a Seq.seq`, if P(`xs`) then P(`Seq.hidel (Seq.Cons x xs)`)

In this task, you will prove that your `look_and_say` doesn't forget any elements: if x is in the input list, then a pair (x, n) is in the output for some n (to make the proof easier, we're not asking you to prove that this n is the correct count).

Define the predicate `x is_ins` ("`x` is an element of the sequence `s`") inductively as follows:

- `x is_in s` if there exists an `xs` such that `(hidel (Cons(x,xs))) ==> s`
- `x is_in s` if there exist `y`,`xs` such that `(hidel (Cons(y,xs))) ==> s` and `x` $\neq$ `y` and `x is_in xs`

Task 2.2 (10%). Prove the following:

Theorem 1. *For all* `x` *and* `s` *and* `s'` *and* `eq`*, if* `x is_in s` *and* `look_and_say eq s ==> s'` *then there exists an* `n` *such that* `(n,x) is_in s'`*.*

using the list-view induction principle for `s`. Hint: Use the fact that `showl (hidel v) ==> v`.

You may use the following lemma about `lasHelp`:

Lemma 1. *For all* `x`,`y`,`s` *and* `n`*,*
if `y` $\neq$ `x` *and* `y is_in s` *and* `lasHelp(s,x,n) ==> s'` *then* `y is_in` s'

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.

```
 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

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| | |
```

Minnie (O) has pieces in spaces $(0, 4)$ and $(1, 3)$. 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. Your estimator must be at least this smart to receive full credit.

  More sophisticated static evaluations 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

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();
```

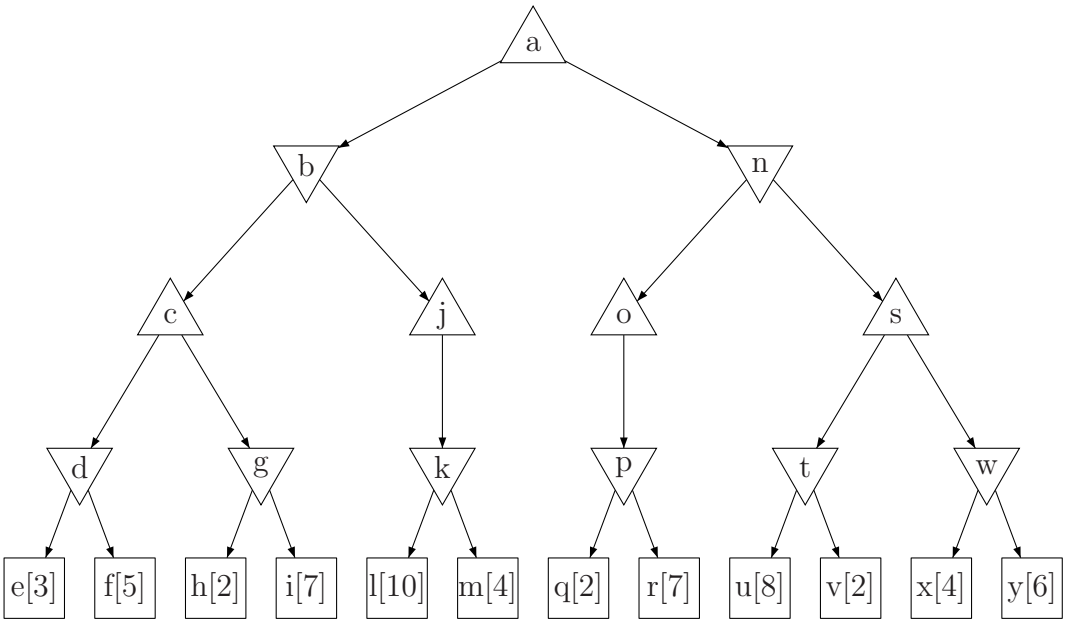
You can write other referee applications to test different variations on the game: different board sizes, different search depths, different game tree search algorithms (such as Jamboree below...).

4 Jamboree

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!

4.1 Alpha-Beta Pruning

Consider the following game tree:



Each node is labeled with a letter, for reference below, and the terminal states are labeled with their values (e.g. given by the estimator). An exhaustive minimax search will conclude that the best move for Maxie is to take the left branch from the root, and its value is 3. However, it may seem like we are doing extra work - isn't there a way to use what we know to reduce the amount of the tree that we have to search? For example, in the right subtree of the root, if Minnie chooses the left branch, the highest possible score will be 2, which is lower than the left subtree's score of 3. Since Minnie can then force a score worse than the left subtree if Maxie chooses the right subtree, we know that Maxie will not choose that move when playing optimally, allowing us to stop searching the rest of the right subtree of the root.

Thus, an optimization presents itself: what if we keep track of the "best" (that is, highest for Maxie, and lowest for Minnie) score that can be guaranteed for each player as we traverse the game tree? That way, we keep more information and are able to prune large portions of

the tree (in the best case). The resulting algorithm is called $\alpha\beta$-*pruning*, because it "prunes" the search tree (cuts off some branches) using two extra values: α, the highest guaranteed score for Maxie, and β, the lowest guaranteed score for Minnie.

Next, we discuss the spec for $\alpha\beta$-pruning. 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`. This values allow the child to pass instructions up to the parent. The standard presentation of pruning is less explicit about what violations of the $\beta > \alpha$ invariant mean, which makes the code harder to read. As usual, we use richer types to communicate better! The meaning of these two messages is determined by the spec for pruning:

> **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 value for that node, AB, satisfies the following:
>
> - If $\alpha < MM < \beta$ then $AB = MM$.
> - If $MM \leq \alpha$ then
>   - $AB =$ `ParentPruned` if s is a Maxie node
>   - $AB =$ `Pruned` if s is a Minnie node
> - If $MM \geq \beta$ then
>   - $AB =$ `Prune` if s is a Maxie node
>   - $AB =$ `ParentPruned` if s is a Minnie node

Roughly, α is what Maxie knows they can achieve, and β is what Minnie knows they 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.

- If the actual minimax value is greater than β, the $\alpha\beta$-pruning value should be `Pruned`. This is because the value is better, for Maxie, than what we already know Minnie can achieve, so Minnie won't make choices that lead to this branch of the tree.

The labels for Minnie are dual.

```
Evaluating state (Minnie,b) with ab=(Pruned,Pruned)
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
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 2: AB-Pruning trace for the above tree

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:*

- To calculate you the value of a node given α and β, you scan across the children of the node, recursively calculating the value of each child. For the first child, α and β are the same as they are for the parent.
- At each step, you use the child's value v to update the value of α *or* the value of β: 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 Maxie can achieve, which can be improved by v). If the value of the child is `ParentPrune`, you stop and label the parent with `Prune`, without considering the remaining children.
- Once all children of a node have been processed, the node is labeled with the final updated α (for Maxie) or β (for Minnie) value—assuming it is within the bounds supplied for the parent node, or with the appropriate `Pruned`/`ParentPrune` if not.

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.

Example Figure 2 shows a trace of this algorithm running on the above tree. As explained below, we use `Pruned` to represent "no bound for α/β." The $c-d-g$ subtree is illustrative:

```
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
```

We start evaluating c with no bounds, so we recursively look at d, and then e, still with no bounds. The estimate is 3, which is trivially within the bounds, so the value 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 the recursive call on f. Since the actual value of f is 5, which is greater than the give β (which is3), and it is a Maxie node, the result is `Pruned`, to signal that the enclosing Minnie

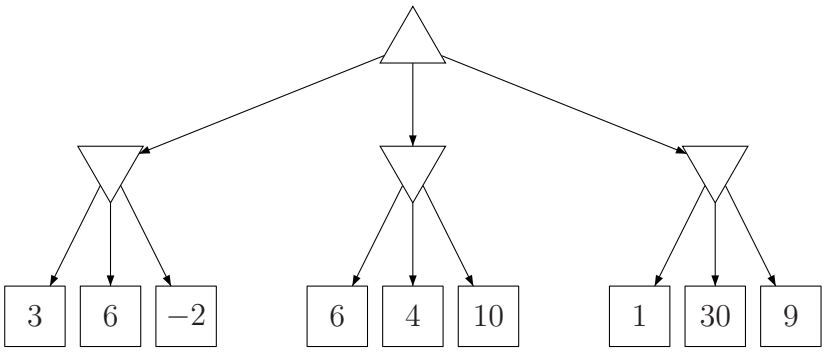
node doesn't want this branch. Since the min of 3 and `Pruned` is still 3, the value of d is 3. Since c is a Maxie node, we update α to be $max(\texttt{Pruned}, 3)$, which is also 3. These bounds are passed down to g and then h. Since the actual value of h is 2, which is less than the given α, and it 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.

4.2 Jamboree

In $\alpha\beta$-pruning, evaluating the children of a node is entirely sequential, because you process the children of a node one-by-one, updating α/β as you go. On the other hand, minimax is entirely parallel: you can evaluate each child in parallel, because there are no dependencies between them. Thus, minimax does more work, but has a better span, whereas pruning does less work, but has a worse span.

The *Jamboree* algorithm manages this tradeoff by evaluating *some* of the children sequentially, updating $\alpha\beta$, and then the remainder in parallel, using the updated information. For this problem, we ask to evaluate the *first child* of each node first, and update $\alpha\beta$ and perhaps prune (as in $\alpha\beta$-pruning), and then evaluate the rest of the children in parallel (as in minimax).

In the above tree, Jamboree will explore the tree in the same way as in Figure 2: 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 3.

4.2.1 Tasks

We have provided starter code in `jamboree.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 datatype:

```
datatype value =
    BestEdge of edge
  | Pruned

datatype result =
    Value of value
  | ParentPrune
```

In some places in the algorithm, the `ParentPrune` option is excluded, and this way we can use the type system to check that the code obeys this invariant.

One such place is in the representation of α and β themselves. We would like to mostly maintain the structure of the traditional $\alpha\beta$-pruning algorithm, where as you go across the children of a node, you max the values into α (for Maxie) or min the values into β (for Minnie). But in our version we want to stop whenever we hit a `ParentPrune`. To make this obvious, we can define α and β to be *values*, rather than *results*: the code *has* to check the pruning condition before getting a real value. Thus, α should be represented by *at most* a `value`, not a `result`. But note that, if α is a value, we need to define $max(\alpha, \texttt{Pruned})$.

Working in the other direction, why not represent α by a `Game.est`, rather than a `value`? At the outside, we start with no bounds, and we need some additional option to signal this. We can also use `Prune` for this purpose, as long as we define what it means for $\texttt{Pruned} < \alpha$.

Now, it would be a total abuse to use the same value `Pruned` to signal these two meanings, except for the fact that they are consistent: for the algorithm, we want $max(\texttt{Pruned}, \alpha) = \alpha$ 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). Thus, both definitions arise from

Jamboree:

```
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 3: Traces for Tree 2

thinking of the type `value` as ordered, with `Pruned` at the bottom, and `BestEdge`'s ordered by the estimates in them.

Dually, for β, we want `Pruned` $\geq \beta$ and $min(\beta, \texttt{Pruned}) = \texttt{Pruned}$, which come from thinking of `Pruned` as the *top* of `value`.

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`, depending on whose turn it is in `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) (v : Game.est) : result = ...
```

that takes an estimate `v` 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) : value = ...
```

`search` may assume the depth is non-zero and the state is `In_play`, and is responsible for evaluating the children according to the Jamboree algorithm and returning the appropriate value. Hint: don't forget that the incoming α/β must be included as a possibility for this value, to get proper pruning (this comes up in the examples above). `evaluate` checks the boundary conditions (depth is 0, game is over) and, if not, searches. `evaluate` must ensure that the `result` satisfies the above **Spec for $\alpha\beta$-pruning**.

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

Testing We have provided a new functor `ExplicitGame` that makes a game from a given game tree, along with two explicit games, `HandoutSmall` (Figure 3) and `HandoutBig` (Figure 2). These should be tested by calling `next_move` on the `start` state with search depth 2 and 4, respectively—an explicit game errors if it tries to estimate in the wrong place. `estimate` prints the states it visits, so you can see what terminals are visited. You may additionally wish to annotate your Jamboree code so that it prints out traces, as above.

5 Extra Credit

These extra credits tasks are 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. Each of these two tasks will be worth somewhere in the vacinity of 25 HW points.

5.1 PopOut

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 5.1 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.

5.2 Parametrized Jamboree

In Jamboree, the choice of using *one* child to update $\alpha\beta$, and doing the rest in parallel, is clearly arbitrary. We can do better:

Task 5.2 In a file `jamboreeec.sml`, implement a functor

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

`prune_percentage` is assumed to be an integer between 0 and 100. For each node, `prune_percentage`% of the children are evaluated sequentially, updating $\alpha\beta$, and the remaining children are evaluated in parallel. For example, with `prune_percentage` = 0, `JamboreeEC` specializes to MiniMax ($\alpha\beta$ are never used), and with `prune_percentage` = 100, `JamboreeEC` specializes to complete, sequential $\alpha\beta$-pruning.