15-150 Fall 2014
Homework 09

Out: Tuesday 11th November, 2014
Due: Tuesday 18th November, 2014 at 23:59 AST

1 Introduction

This homework will allow you to gain practice working with complex control flows both in a sequential and parallel setting. Specifically, you will be writing code that unfolds a potentially large search tree and developing heuristics to do so effectively. This will give you experience working with sequences and tree search.

The startup code for this homework comprises numerous files, although you will need to edit only a few. It will reinforce the experience, very common in practice, of working with a large body of code written by others. In fact, several programmers contributed to the starter code, which means that style and polish vary. You will have to read quite a bit of documentation, sometimes not written as precisely as you wish. Some of that documentation will not be in this file, but in various signatures. In fact, you will need to do a lot of figuring out on your own. How do you like this for a real-world experience?[1]

1.1 Getting the Homework Assignment

The starter files for the homework assignment have been distributed through Autolab at

`https://autolab.cs.cmu.edu`

Select the page for the course, click on "hw09", and then "Download handout". Uncompressing the downloaded file will create the directory `hw09-handout` containing the starter files for the assignment. The directory where you will be doing your work is called `hw/09` (see below): copy the starter files there.[2]

[1] Well, this is actually a lot better than the real world: the documentation is pretty complete and nothing it says is wrong. Moreover we are giving you a lot of hints throughout the homework. Oh wait, that's what they always say!

[2] The download and working directory have different names so that you don't accidentally overwrite your work were the starter files to be updated. **Do not do your work in the download directory!**

1.2 Submitting the Homework Assignment

Submissions will be handled through Autolab, at

`https://autolab.cs.cmu.edu`

In preparation for submission, your `hw/09` directory will need to contain the files `las.sml`, `connect4.sml`, `run.sml`, `alphabeta.sml`, `jamboree.sml`, and `test-all.sml` with your code. The starter code for this assignment contains incomplete versions of these files. You will need to complete them with the solution to the various programming tasks in the homework.

To submit your solutions, run

```
make
```

from the `hw/09` directory on any Unix machine (this include Linux and Mac). This will produce a file `hw09-handin.tgz`, containing the files to be handed in for this homework assignment. Open the Autolab web site, find the page for this assignment, and submit your `hw09-handin.tgz` file via the "Handin your work" link. If you are working on AFS, you can alternatively run

```
make submit
```

from the `hw/09` directory. That will create `hw09-handin.tgz`, containing the files to be handed in for this homework assignment and directly submit this to Autolab.

All submission will go through Autolab's "autograder". The autograder simply runs a series of tests against the reference solution. Each module has an associated number of points equal to the number of functions you need to complete. For each such function, you get 1.0 points if your code passes all tests, and 0.0 if it fails at least one test. Click on the cumulative number for a module for details. Obtaining the maximum for a module does not guarantee full credit in a task, and neither does a 0.0 translate into no points for it. In fact, the course staff will be running additional tests, reading all code, and taking into account other aspects of the submitted code such as structured comments and tests (see below), style and elegance.

To promote good programming habits, your are limited to a maximum of 6 submissions for this homework. Use them judiciously! In particular, make sure your code compiles cleanly before submitting it. Also, make sure your own test suite is sufficiently broad.

The SML files `las.sml`, `connect4.sml`, `run.sml`, `alphabeta.sml`, `jamboree.sml`, and `test-all.sml` 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.

1.3 Due Date

This assignment is due on Tuesday 18th November, 2014 at 23:59 AST. Remember that there are no late days for this course and you will get 2 bonus points for every 12 hours that you submit early.

1.4 Code Structure

On this and future assignments, we will be grading your programs on more than just their input-output behavior. It's not enough to have programs that happen to work: they need to clearly state what they do, have some empirical evidence that they work as advertised, and be easy for other people to read and reason about.

You must use the following five step methodology for writing functions, for *every* function you write in this assignment:

1. In the first line of comments, write a call template of the function.
2. In the second line of comments, specify via a `REQUIRES` clause any assumptions about the arguments passed to the function.
3. In the third line of comments, specify via an `ENSURES` clause what the function computes (what it returns).
4. Implement the function (include type annotations for the arguments and result of the function). Your code should compile without errors (and preferably without warnings). Compilation errors in a file will result in every task carried out in this file getting no credit.
5. Provide test cases, generally in the format
   `val <return value> = <function> <argument value>`.
   Test cases are individual. Sharing them is not allowed.

For example, for the factorial function presented in lecture:

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

(* Tests: *)

val 1 = factorial 0
val 720 = factorial 6
```

1.5 Testing Modules

Because modules encourage information hiding, the way to test SML structures and functors is a bit different from what you did in the past. In fact, outside of a module, you may have no way to view the values of an abstract type. This means you can't compare the result of an operation with the expected value because you have no way to construct this expected value.

So, how to test modular code? There are essentially two ways to proceed.

Inside-the-box testing: You can't build values outside your module, but you can do so inside (typically). Then, what you would do is to put your normal tests inside the structure you are working on. As usual, if a test fails, a `binding non exhaustive` exception will be raised.

This is a bit trickier to do with functors, because you may not have a way to build values that depend on the functor's parameters. In this case, outside-the-box testing is your only option.

Outside-the-box testing: Many modules export a printing function and an equality function (conventionally called `toString`) and `eq`, respectively). You can then use the equality function to test that the value returned by a function is the value you expect. You can use the printing function to visualize returned values of hidden type.

When a module does not provide such functions, it exports operations that interact with each other, somehow. You can leverage these interactions for testing purposes. For example, a dictionary exports `insert` and `lookup` operations. You may test a module implementing dictionaries by populating a dictionary using `insert` and then use `lookup` to check that the expected entries are in it (and that unexpected entries are not).

Best of all, you want to use a combination of inside- and outside-the-box testing. Notice that inside-the-box testing is implementation-dependent, but outside-the-box is not.

1.6 Style

Programs are written for people to read — it's convenient that they can be executed by machines, but their high level text is a way for one person to explain an idea to another person. Your code should reflect this, and we will grade your code on how easy it is to understand.

The published style guide is your primary resource here. Strive to write clear, concise, and elegant code. If you have any questions about style, or just have a feeling that a piece of code could be written more simply, don't hesitate to ask!

1.7 The Compilation Manager

We will be using several SML files in this assignment. In order to avoid tedious and error-prone sequences of `use` commands, we will use SML's *compilation manager*. The compilation manager (CM) is a system that keeps track of what files have been modified and runs just them (and the files that depend on them) through SML. If you have used `make` on a Unix system, the idea is very similar.

Using CM is simple. In fact, there are two ways to do so:

Go to the directory containing your work and run at the terminal prompt (written `#`):

```
# sml -m sources.cm
```

or

Launch SML from the directory containing your work, and then run at the SML prompt (written `-`):

```
- CM.make "sources.cm";
```

Both wil load all the files listed in `sources.cm`[3] and take you to the SML prompt. Do so whenever you change your code. No need to call `use` — in fact you may confuse CM. For large programs, CM offers a better interface to the command line. There is less typing and less of an issue with name shadowing between iterations of your code.

In short, on this assignment, the development cycle will be:

1. Edit your source files.
2. Type either `sml -m sources.cm` at the terminal prompt or `CM.make "sources.cm";` at the SML prompt.
3. Fix errors and repeat.

`CM.make` creates a directory called `.cm` in the current working directory. It gets populated with metadata needed to work out compilation dependencies. The `.cm` directory can safely be deleted at the completion of this assignment (in fact, it can become quite large)

It's sometimes happens that the metadata in the `.cm` directory gets into an inconsistent state — if you run `CM.make` with different versions of SML in the same directory, for example. This often results in bizarre error messages. When that happens, it is safe to delete the `.cm` directory and compile again from scratch.

[3] The file `sources.cm` contains a list of the files tracked by CM. Feel free to take a peek if you are curious!

1.8 Common Sequence Functions

For your convenience, here is a brief description of some of the functions on sequences.

- `Seq.length: 'a Seq.seq -> int`
  `Seq.length s` returns the number of elements in the sequence `s`.
- `Seq.nth: int -> 'a Seq.seq -> 'a`
  `Seq.nth i s` returns the element of sequence `s` at index `i` (starting from 0), assuming it is in bounds.
- `Seq.tabulate: (int -> 'a) -> int -> 'a Seq.seq`
  `Seq.tabulate f n` computes a sequence of length `n` such that the value of each element of the sequence is the result of applying the function `f` to its index.
- `Seq.map: ('a -> 'b) -> 'a Seq.seq -> 'b Seq.seq`
  `Seq.map f s` returns a sequence whose elements are the result of applying the function `f` to the corresponding element in the sequence `s`.
- `Seq.reduce: ('a * 'a -> 'a) -> 'a -> 'a Seq.seq -> 'a`
  `Seq.reduce g e s` combines all the elements of the sequence `s` using the binary function `g` and base value `e`.
- `Seq.mapreduce: ('a->'b) -> 'b -> ('b * 'b -> 'b) -> 'a Seq.seq -> 'b`
  `Seq.mapreduce f e g s` chains the functionalities of `Seq.map` and `Seq.reduce` by applying the function `f` to each element of the sequence `s` before combining them as in `Seq.reduce` with `g` and `e`.
- `Seq.filter: ('a -> bool) -> 'a Seq.seq -> 'a Seq.seq`
  `Seq.filter f s` returns that sequence obtained by keeping only the elements of `s` on which `f` returns `true`.
- `Seq.empty: unit -> 'a Seq.seq`
  `Seq.empty ()` returns the empty sequence.
- `Seq.singleton: 'a -> 'a Seq.seq`
  `Seq.singleton x` returns the one-element sequence containing the value of `x`.
- `Seq.append: 'a Seq.seq -> 'a Seq.seq -> 'a Seq.seq`
  `Seq.append s1 s2` returns the sequence obtained by appending sequence `s2` to the end of sequence `s1`.
- `Seq.flatten: 'a Seq.seq Seq.seq -> 'a Seq.seq`
  `Seq.flatten S` returns the sequence obtained by appending all sequences in `S`.

Function	Inputs	Work	Span	Note
`Seq.length s`	n = size of `s`	$O(1)$	$O(1)$	
`Seq.nth i s`	n = size of `s`	$O(1)$	$O(1)$	
`Seq.tabulate f n`		$O(n)$	$O(1)$	[1]
`Seq.map f s`	n = size of `s`	$O(n)$	$O(1)$	[2]
`Seq.reduce g e s`	n = size of `s`	$O(n)$	$O(\log n)$	[2]
`Seq.mapreduce f e g s`	n = size of `s`	$O(n)$	$O(\log n)$	[2]
`Seq.filter f s`	n = size of `s`	$O(n)$	$O(\log n)$	[2]
`Seq.empty ()`		$O(1)$	$O(1)$	
`Seq.singleton x`		$O(1)$	$O(1)$	
`Seq.append s1 s2`	n_1 = size of `s1` n_2 = size of `s2`	$O(n_1 + n_2)$	$O(1)$	
`Seq.flatten s`	n = sum of sizes of `s`	$O(n)$	$O(\log n)$	

Notes:

[1] assuming $O(1)$ work and span for `f`.

[2] assuming $O(1)$ work and span for `f`. Multiply by $W_{\texttt{f}}(n)$ or $S_{\texttt{f}}(n)$ if work and span of `f` is independent of its input, but dependent on n. Same thing for `g`.

Printing Sequences

To print a sequence, you can use the function `Seq.toString: ('a -> string) -> 'a Seq.seq -> string`. The first argument is a function that prints the individual elements of the sequence. For example,

```
Seq.toString Int.toString (Seq.tabulate (fn i => i+1) 12)
```

Testing with Sequences

To test code returning sequences, you can use the function `Seq.eq: ('a * 'a -> bool) -> ('a Seq.seq * 'a Seq.seq) -> bool`. The first argument is a function that tests the equality of pairs of elements. For example,

```
Seq.eq (op=) (Seq.tabulate (fn i => i+1) 12, Seq.tabulate (fn j => 1+j) 12)
```

Conversion to Lists

The functions `Seq.toList: 'a Seq.seq -> 'a list` and `Seq.fromList: 'a list -> 'a Seq.seq` allow converting to and from a list. Their span is $O(n)$ where n is the size of the input sequence or list. They should therefore be avoided except for actual list conversion.

2 Views

In this assignment, 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
```

A *view* of an abstract type is a way to observe it in a particular concrete form, even though you don't know how it's actually implemented. Consult the posted lecture notes for a more detailed introduction to view types.

Task 2.1 (6 points) Use the list view of sequences to write the function

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

in file `las.sml` and test it in `test-all.sml`. This function generalizes look-and-say from lists of integers you wrote in an earlier homework to sequences of any element type. For example, if `streq` is a function that tests strings for equality:

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

You should refer to Homework 3 for a good way to structure this solution. Do `not` use `Seq.nth`.

This function and the general technique of using views should be helpful to you throughout the assignment.

3 Games

In this homework you will implement a specific game and write a generic game player. There are two major parts: writing the representation of the game, and writing both a sequential and parallel version of the popular $\alpha\beta$-pruning algorithm.

A *game* G between a player M (or Maxie) and an opponent m (or Minnie) is mathematically described as a quadruple

$$G \quad = \quad (S_M \cup S_m, s_0, F_M \cup F_m, \delta_M \cup \delta_m)$$

where

- S_M is the set of *states* of Maxie and S_m is the set of states of Minnie,
- $s_0 \in S_M$ is the *initial state* — Maxie always starts,
- $F_M \subseteq S_M$ is the set of *final states* of Maxie and $F_m \subseteq S_m$ is the set of final states of Minnie,
- $\delta_M : S_M \to \mathcal{P}(S_m)$ and $\delta_m : S_m \to \mathcal{P}(S_M)$ are the *transition functions* of Maxie and Minnie respectively. They describes the legal moves of the game. The first associates to each state $s \in S_M$ in which it is Maxie's turn to play the set of Minnie's states resulting from performing a valid move from s. The second is dual.

Any game can be described in this way.[4]

3.1 Implementing Games

Implementing a game amounts to expressing the above definitions in a programming language, SML in our case. We need to choose a data structure that represents the states of each player, define the initial state, decide on which states are final, and implement the transition function. At its core, this is what the `GAME` signature in file `game.sig` provides. Note that it is a lot more explicit than the mathematical definition: among other things, the type `player` gives names to the players, and the type `move` provides a direct way to describe moves. This entails a richer set of functions than just the transition functions δ_M and δ_m.

This signature also contains numerous additional declarations to help debugging and make playing the game easier. One last thing it provides is a utility function, `estimate`, together with a type `est` that accounts for the status of the game.

To see `GAME` in action, check the file `nim.sml` which implements the Nim game.

[4]The specific type of games captured by this definition are called two-player, deterministic, perfect-information, finite-branching and zero-sum games.

3.2 Connect 4

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

For example, Figure 1 shows a few example boards and the moves from one to the other. In the board at the top of the figure, 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. It is Maxie's turn, and she decides to drop a piece in column 0, which falls to the bottom of column 0. The resulting board is shown in the middle of the figure. It is now Minnie's turn. Note that Maxie has three pieces in a row, and could win by playing in column 1 if it is still free on her next turn. Minnie notices this and blocks Maxie by placing a piece in column 1. The resulting board is shown at the bottom of the figure.

3.3 Implementing Connect 4

The support code for this homework includes the `GAME` code discussed in the lecture notes (see course web page). Part of an implementation of the `GAME` signature for Connect 4 is provided in the starter file `games/connect4/connect4.sml`. It uses the following representation of game states:

```
datatype position = Filled of player | Empty

(* (0,0) is bottom-left corner *)
datatype c4state = Unimplemented (* use in starter code only *)
                 | 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 `lib/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 is thought of as the origin, so matrix indices `(x,y)` can be thought of as xy-coordinates in the plane. So, in the following board:

```
 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, please type your move: 1
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


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

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

Task 3.1 (0 points) Play Connect 4 on paper with a friend. (No, really.)

Task 3.2 (50 points) Complete the implementation of the `Connect4` functor in `connect4.sml` and test it in `test-all.sml`. This takes as parameters two values specifying the dimensions of the board — yes, functors can be passed all kinds of arguments. Here are some remarks that may be useful:

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

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

  which you will need to implement. If `r` is the index of the lowest free row in column `i`, then `lowestFreeRow s i` $\cong$ `SOME(x)`.

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

- You may find `lookAndSay` helpful for your implementation.

- `estimate` is pretty open-ended. A very naïve estimator is to award some number of points for each run of pieces 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 needs to be at least this clever.

  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 correctly implement the game strategies in the next section than to spend time improving your estimator.

3.4 Running Your Game

The file `game/connect4/run.sml` uses the `Referee` to construct a Connect 4 match, with Maxie as a human player (that's you!) 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 Playing a Game

A *match* $\bar{s}$ is a series of states obtained by applying legal moves. Therefore,

$$\bar{s} \;=\; (s_0, s_1, \ldots, s_n)$$

such that $s_{2i+1} \in \delta_M(s_{2i})$ and $s_{2i+2} \in \delta_m(s_{2i+1})$ for all $0 \leq i < n$. The match starts at the initial state s_0 and ends at s_n. A *winning match* $\bar{w}$ for Maxie (for Minnie) is a match that ends in a state in F_M (in F_m) and does not contain any other final states. In symbols:

$$\bar{w} \;=\; (s_0, s_1, \ldots, s_n)$$

where $s_n \in F_M$ ($s_n \in F_m$) and $s_i \notin F_M \cup F_m$ for $i < n$ (and of course$s_{2i+1} \in \delta_M(s_{2i})$ and $s_{2i+2} \in \delta_m(s_{2i+1})$ for all $0 \leq i < n$). A *draw* is a match $(s_0, s_1, \ldots, s_n)$ such that s_n is not a winning state for either Maxie nor Minnie ($s_n \notin F_M \cup F_m$) and there is no transition from it (i.e., $(\delta_M \cup \delta_m)\ s_n = \emptyset$).

The goal of playing a game for Maxie (for Minnie) amounts to finding a winning match for Maxie (for Minnie). To do so, we need to explore the *search tree* defined by recursively applying all valid moves starting from the initial state.

A *strategy* is a way to choose the next move when playing the game. Mathematically, it is just a pair of functions $\sigma_M : S_M \to S_m$ and $\sigma_m : S_m \to S_M$, that, given a state for Maxie (for Minnie), returns some "preferred" next state (if any).

4.1 Implementing Players

A *player* is a procedure that explores the search tree of a game according to a strategy. The signature `PLAYER`, in file `players/player.sig`, is a generic interface for players. Structures ascribing to `PLAYER` implement players that employ a specific strategy.

The `PLAYER` signature defines a single function, `next_move`, which selects the move to be played from any given state according to the specific strategy being implemented.

4.2 Players for Specific Strategies

Two structures ascribing to `PLAYER` are given to you: one is MiniMax (in file `players/minimax.sml`) and the other is a human player (in `players/humanplayer.sml`) which simply asks the user for what the next move should be. You can experiment with them using the Nim game (in directory `games/nim`) and also the specific game you are asked to implement in this homework.

You are asked to implement two more strategies ascribing to `PLAYER`: one is a version of $\alpha\beta$-pruning, and the other is Jamboree, a parallel variant of $\alpha\beta$-pruning.

4.3 Alpha-Beta Pruning

In the MiniMax game tree search algorithm, each player chooses the move that, assuming both players make optimal moves, 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 simple tree shown in Figure 2

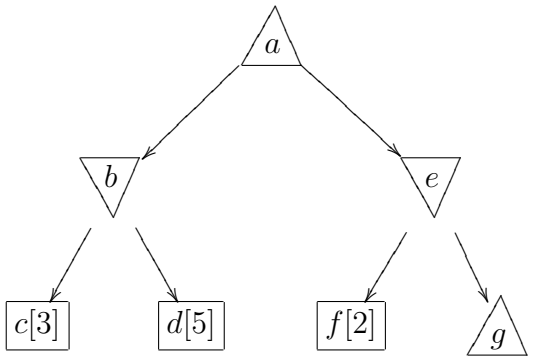


Figure 2: Simple Game Tree

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 Maxie nodes and are also labeled with their utility values (given by the game's `estimate` function).

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 long time 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.

4.3.1 Setup

In $\alpha\beta$-pruning, we keep track of two values α and β, 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. Specifically, 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`, signaling 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 invariants:
>
> - 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 lead 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 $\texttt{Pruned} \leq \alpha$ (`Pruned` is no bound at all, so anything is better than it). Dually, for β, we want `Pruned` to be the *biggest* β: $\texttt{Pruned} \geq \beta$ and $min(\beta, \texttt{Pruned}) = \beta$.

4.3.2 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.

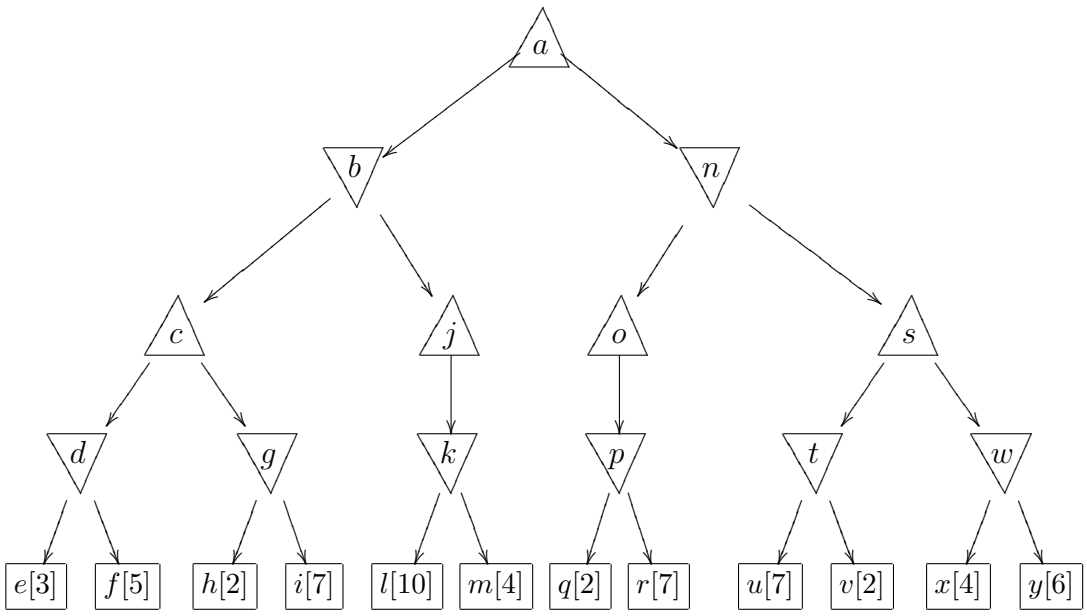


Figure 3: Extended $\alpha\beta$-pruning example

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.

4.3.3 Extended Example

Figure 4 shows a trace of this algorithm running on the tree in Figure 3. As explained above, we use `Pruned` in the initial call to represent "no bound for α/β." The starting at node c is isomorphic to the example tree in Figure 2. We will show how the algorithm runs on it — see Figure 4 for how it processes the whole tree in Figure 3. 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)

Evaluating state (Minnie,b) 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 4: AB-Pruning trace for the tree in Figure 3

4.3.4 Tasks

We have provided starter code in `games/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 over 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 `result`s, 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 `value`s.

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 points) Define the function

```
updateAB: Game.state -> alphabeta -> value -> alphabeta
```

such that `updateAB s ab v` 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 points) Define the function

```
value_for: Game.state -> alphabeta -> value
```

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

Task 4.3 (4 points) Define the function

```
check_bounds: alphabeta -> Game.state -> Game.move -> Game.est -> result
```

such that `check_bounds (alpha,beta) state incomingMove e` 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 points) Define the mutually recursive functions:

```
evaluate: int -> alphabeta -> Game.state -> Game.move         -> result
search:   int -> alphabeta -> Game.state -> Game.move Seq.seq -> value
```

with call patterns `evaluate search depth ab s incomingMove` and `search depth ab s moves`. Here, `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`.[5] **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 3, check for this bug.**

The function `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 points) Define `next_move`.

Task 4.6 (2 points) In `games/connect4/run.sml`, use the `Referee` to define a structure `C4_HvAB` that allows you to play Connect 4 against your $\alpha\beta$-pruning implementation.

[5] 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.

Testing: Test your functions individually in `test-all.sml`. Additionally, you can test your overall implementation by using the `ExplicitGame` functor. This functor makes a game from a given game tree, along with two explicit games, `HandoutSmall` (Figure 5) and `HandoutBig` (Figure 4). These can be used for testing as follows:

```
- structure TestBig = AlphaBeta(structure G = HandoutBig
                                val search_depth = 4);
- 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(structure G = HandoutSmall
                                val search_depth = 2);
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 produces 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.

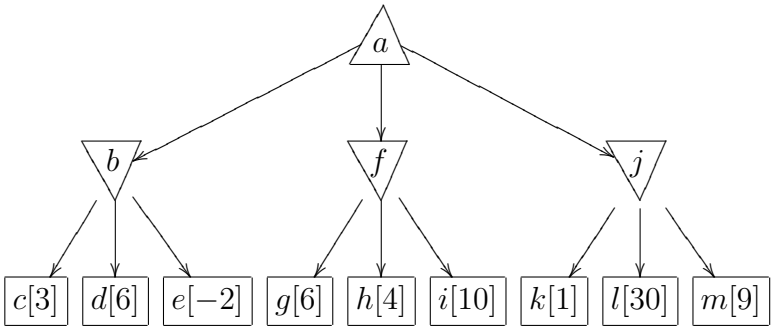
5 Jamboree

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 trade-off*: 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 trade-off 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.

You will complete the code in the file `players/jamboree.sml`. The functor therein takes one more parameter than the code for $\alpha\beta$-pruning, `prune_percentage: real`. it 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 3, Jamboree will explore the tree in the same way as in Figure 4: 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 5 for the traces.

5.1 Tasks

Test all the code below in `test-all.sml`.

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 points) Define the functions:

```
evaluate: int -> alphabeta -> Game.state -> Game.move          -> result
search:   int -> alphabeta -> Game.state -> Game.move Seq.seq
        -> Game.move Seq.seq -> value
```

using the Jamboree algorithm.

The spec for `evaluate search depth ab s incomingMove` 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.

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

```
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 5: Traces for Tree 2

The call `search depth ab s abmoves mmoves` 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`.

Note that while the Jamboree algorithm is a hybrid of $\alpha\beta$ and MiniMax — and the implementations of those two will very much inform the implementation of Jamboree — it is not the same as either. In particular, it will not help you to try to call the search and evaluate from either MiniMax or $\alpha\beta$ directly in your Jamboree implementation.

Task 5.3 (2 points) Define `next_move`.