COMP 105 Homework: Standard ML Modules

Due Tuesday, May 3 at 11:59 PM

The purpose of this assignment is threefold:

- To get some practice with Standard ML modules
- To use Standard ML modules to put together a nontrivial program.
- To see how to reuse code that depends not just on other values, but also on other types.

You will complete problems 1–2 and problems A and C. (There was a problem B, but I took it out.)

The code in this handout, together with a `compile` script, can be had by

```
git clone linux.cs.tufts.edu:/comp/105/git/ttt
```

ML Modules finger exercises

1. **A simple functor (Difficulty *, Time *)**. On March 7, Greg Morrisett gave a guest lecure on Haskell type classes. He talked about QuickCheck and how it generates random data. This capability can be duplicated using ML modules.
   a. *Define a signature* `ARBITRARY` with a type `t` and an operation `arbitrary` that generates random data of type `t`.

      The `arbitrary` operation should be similar to the `arby` operation in the lecture notes under the title "What is Testable?" But there are a couple of differences:

      - The `arbitrary` operation should not be polymorphic: it should always return a value of type `t`.
      - Because ML permits side effects, it is not necessary or desirable to supply a source of random numbers. Instead, the `arbitrary` operation can take an empty tuple as argument.

   b. *Write a functor* `ArbitraryList` that takes one argument `A` with signature `ARBITRARY` and produces a new structure that *also* has signature `ARBITRARY`, but in which the `arbitrary` operation produces an arbitrary *list* of type `A.t list`. You will need to come up with appropriate definitions of both `t` and `arbitrary`.

      Hints:

      ◊ Don't overlook http://www.standardml.org/Basis, especially `List.tabulate`. (You can see an example in the same lecture notes.)

      Also try `help "lib";` in Moscow ML; you may find `Random.range` useful.

      ◊ If you want to test your functor, define a structure `ArbitraryInt : ARBITRARY where type t = int`.

   Put your signature and functor in file `arbitrary.sml`.
2. **Data structures (Difficulty *, Time **)**. A *heap* is a collection of *elements* with an operation that quickly finds and removes a minimal element. The elements of a heap are therefore totally ordered.
   a. *Design an abstraction* for representing heaps in ML. Your abstraction may be mutable or immutable.

      Formalize your abstraction by giving an ML signature `HEAP` describing the abstraction. Be sure to

      ◊ Define two **abstract** types: one to represent a heap and one to represent an element.
      ◊ Identify each operation as a creator, producer, mutator, or observer
      ◊ Specify what each operation does, either using informal English, algebraic laws, or both

      The elements of a heap are totally ordered; be sure to **expose that total order in the interface**.

Put your signature into a file called `heap-sig.sml`. You will need to compile it with the `-toplevel` option, e.g.,

```
mosmlc -toplevel -c heap-sig.sml
```

Notice that for this part of the problem you write **no code**. All you write is the interface.

b. *Use your abstraction* to implement heap sort. That is, write a functor that takes a structure matching signature `HEAP` and produces a structure that contains a function that sorts a list of elements by inserting all the elements into a heap, then removing them one by one until the heap is empty.
   - ◊ Give your functor an explicit result signature, paying careful attention to type revelation.
   - ◊ *You need not implement `HEAP`*. This is the whole point!

Put your functor into a file called `heapsort.sml`. You will need to compile it with the `-toplevel` option, e.g.,

```
mosmlc -toplevel -c heap-sig.ui heapsort.sml
```

Because the `heapsort.sml` refers to signature `HEAP`, you must pass it the `heap-sig.ui` file where signature `HEAP` is defined. You will have produced `heap-sig.ui` by compiling `heap-sig.sml`.

Playing Adversary Games: Overview

In problems A-C below, you will implement and use a system for playing simple adversary games. The program will show game configurations, accept moves from the user and choose the best move.

The system is based on an abstract game solver (AGS) which, given a description of the rules of the game, will be able to select the best move in a particular configuration. An AGS is obtained by abstracting (separating) the details of a particular game from the details of the solving procedure. The solving procedure uses exhaustive search: it tries all possible moves and picks the best. Such a search can solve games of complete information, provided the configuration space is small enough. And the search is general enough that we can abstract away details of many games, separating the implementation of the solver from the implementation of the game itself.

To separate game from solver, in such a way that a single solver can be used with many games, requires a carefully designed interface. In this problem, we give you such an interface, which is specified using the SML signature `GAME`. (The signature was designed by George Necula and modified by Norman Ramsey.)

The `GAME` signature declares all the types and functions that an Abstract Game Solve must know about a game. The signature is general enough to cover a variety of games. Even details like ``the players take turns'' are considered to be part of the rules of the game—such rules are hidden behind the `GAME` interface, and the AGS operates correctly no matter what order players move in. (You could even implement a solitaire as a ``two-player'' game in which the second player never gets a turn!)

You will use two-player games in the last two parts of this assignment: implement a particular game and implement an AGS of your own.

The idea behind the Abstract Game Solver (AGS)

As players move, the state of a game moves from one *configuration* to another. In any given configuration, our solver considers all possible moves. After each move, it examines the resulting configuration and tries all possible moves from that configuration, and so on. In~each configuration, the solver assumes that the player plays perfectly, that is, whenever possible the player will choose a move that forces a win.

This method (``exhaustive search'') is suitable only for very small games. Nobody would use it for a game like chess, for example. Nevertheless, variations of this idea are used successfully even for chess; the idea is to stop or ``prune'' the search before it goes too far.

Basic data in the problem: Players and outcomes

Representation is the essence or pgoramming. We~start by describing basic representations for the essential facts we assume about each game:

3. There are two *players*.
4. A game ends in an *outcome*: either one of the players has won, or the outcome is a tie.

The representations of these central concepts are *exposed*, not abstract. They are given by the signature `PLAYER`.

<player-sig.sml>=
```
signature PLAYER = sig
  datatype player  = X | O    (* 2 players called X and O *)
  datatype outcome = WINS of player | TIE

                              (* Returns the other player *)
  val otherplayer : player -> player

  val toString    : player -> string

  val outcomeToString : outcome -> string
end
```

> Defines `otherplayer`, `outcome`, `outcomeToString`, `PLAYER`, `player`, `toString` (links are to index).

The signature `player` also includes some functions that compute with players and outcomes. Here's the implementation of signature `PLAYER` in a structure called `Player`.

<player.sml>=
```
structure Player :> PLAYER = struct
  datatype player  = X | O
  datatype outcome = WINS of player | TIE

  fun otherplayer X = O
    | otherplayer O = X

  fun toString X = "X"
    | toString O = "O"

  fun outcomeToString TIE = "Tie"
    | outcomeToString (WINS p) = toString p ^ " wins"
end
```

> Defines `otherplayer`, `outcome`, `outcomeToString`, `Player`, `player`, `toString` (links are to index).

Although it might seem overly pedantic, we prefer to isolate details like the player names and how to convert them to a printable representation. To refer to `Player` types, constructors, and functions, you will use the ``fully qualified" ML module syntax, as in the examples `Player.otherplayer p`, `Player.X`, `Player.O`, and `Player.WINS p`. The last three expressions can also be used as patterns.

Specification of a Game

The AGS can play any game that meets the specification given in signature `GAME`. This signature gives a contract for an entire module, which subsumes the contracts for all its exported functions.

<game-sig.sml>=
```
signature GAME = sig
  structure Move : sig  (* information related to moves *)
```

```
    eqtype move              (* A move (perhaps a set of coordinates) *)
    exception Move           (* Raised (by makemove & fromString) for invalid moves *)
    val fromString : string -> move
                (* converts a string to a move; If the string does not
                   correspond to a valid move, fromString raises Move *)
    val prompt : Player.player -> string
                             (* Given a player, return a request for a move
                                for that player *)
    val toString : Player.player -> move -> string
                             (* Returns a short message describing a
                                move. Example: "Player X moves to ...".
                                The message may not contain a newline. *)
  end

  type config     (* A representation for a game configuration. It
                     must include a full description of the state
                     of a game at a particular moment, including
                     keeping track of whose turn it is to move.
                     Either configurations must be immutable,
                     or if they are mutable, it must be impossible
                     for a client to tell that a mutation has
                     taken place. *)

  val toString : config -> string
                  (* Returns an ASCII representation of the
                     configuration.  The string must show whose turn it is. *)

  val initial : Player.player -> config
                  (* Initial configuration for a game when
                     "player" is the one to start.  We need the
                     parameter because the configuration includes
                     the player to move. *)

  val whoseturn  : config -> Player.player
                  (* Extracts the player whose turn is to move
                     from a configuration. We need this function because
                     the solver may need to know whose
                     turn it is, and the solver does not have
                     access to the representation of a configuration.
                   *)

  val makemove: config -> Move.move -> config
                  (* Changes the configuration by making a move.
                     The player making the move is encoded in the
                     configuration.  Be sure that the new
                     configuration knows who is to move. *)

  val outcome : config -> Player.outcome option
                  (* If the configuration represents a finished game,
                     return SOME applied to the outcome.
                     If the game isn't over, return NONE. *)

  val finished : config -> bool
                  (* True if the configuration is final. This
                     might be because everybody is stuck (Tie) or
                     because one has won *)

  val possmoves : config -> Move.move list
                  (* A list of possible moves in a given
                     configuration. ONLY final configurations
                     might return nil.  This means that a
                     configuration which is not final MUST have
                     some possible moves.  In other words,
                     part of the contract is that if 'finished cfg'
                     is false, 'possmoves cfg' must return non-nil. *)

end
```

Defines `config`, `finished`, `GAME`, `initial`, `makemove`, `Move`, `outcome`, `possmoves`, `toString`, `whoseturn` (links are to index).

This is a broad interface. For example, there are three different ways to tell if a game is over!

Compiling Standard ML modules using Moscow ML

To compile an individual module using Moscow ML, you type

```
mosmlc -c -toplevel filename.sml
```

This puts compiler-interface information into *`filename`*`.ui` and implementation information into *`filename`*`.uo`. Perhaps surprisingly, either a signature or a structure will produce *both* `.ui` and `.uo` files. This behavior is an artifact of the way Moscow ML works; it should not alarm you.

Once you have compiled a bunch of modules, you can make an executable binary using `mosmlc`. Here is an example of a command line I use on my system to build an interactive game player:

```
  mosmlc -toplevel -o games player-sig.uo player.uo game-sig.uo \
                            ags-sig.uo play-sig.uo slickttt.uo ttt.uo \
                            ags.uo aggress.uo nim.uo four.uo peg.uo mrun.uo
```

Order does matter here; for example, I have to put `player.uo` *after* `player-sig.uo` because the `Player` structure defined in `player.sml` uses the `PLAYER` signature defined in `player-sig.sml`.

The `git` repository for this assignment includes a `compile` script that may help with compiling Moscow ML modules.

When you are debugging, it is tremendously useful to get compiled modules into the interactive system. You do this with the Moscow ML `load` function. I have an example use of load in Part B. **If you recompile, you must exit Moscow ML and start over**. Once a module is loaded, loading it again has no effect, even if the code has changed.

Implement Tic-Tac-Toe

**A.Implement the description for ``Tic-Tac-Toe.'' (Difficulty \*\*, Time \*\*\*)** More precisely, implement a module `TTT` matching signature `GAME` that describes Tic-Tac-Toe. If you are unfamiliar with Tic-Tac-Toe (elsewhere called ``Noughts and Crosses'', you can find an explanation at the end of this assignment. Call your structure `TTT`, put it in the file `ttt.sml`, and use the following pattern :

<template for `ttt.sml`>=
```
structure TTT :> GAME =
    struct
        structure Move = struct
          type move = ... (* or use a datatype *)
          exception Move
          ...
        end

    type config = ...  (* or use a datatype config = *)


    fun initial p = ...
    fun whoseturn c = ...

        ... and so on for all the values in GAME ...
    end
```

Defines `TTT` (links are to index).

Note the use of `:>`, which means that the *only* access to the types is through the functions in the `GAME` signature.

When writing `TTT`, you must define *all* types and values mentioned in the signature `GAME`, and all values must have the types specified. You might want to define additional values, which you will be able to use as helper functions. These functions cannot be called from anyone else's code: because `TTT` is forced to have signature `GAME`, the functions are not visible outside the `TTT` module, and therefore no other code can depend on them.

So we can test your code, we insist that you use the following names of squares in `Move.toString` and `Move.fromString`:

```
 upper left  | upper middle  |   upper right
-------------+---------------+---------------
middle left  |    middle     |  middle right
-------------+---------------+---------------
 lower left  | lower middle  |   lower right
```

You should always print and recognize these full names. If you wish, you may also recognize the abbreviations `ul`, `um`, `ur`, `ml`, `m`, `mr`, `ll`, `lm`, and `lr` in the function `Move.fromString`.

Here are some hints about how to get started.

a. Choose how you will represent the state of the game (i.e., define `config`). This step is crucial because it determines how complex your implementation will be. There are many possible representations; any one is OK provided you are able to implement the functions required by the signature. Choose a representation that will make it easy to implement `makemove`, `possmoves`, and `outcome`.

   The AGS cannot possibly depend on your choice of representation (the ML module system guarantees it), so you are free to choose whatever representation you like. Even more important, **you can change your reprsesentation at any time**, and no code outside your own module will be affected. If you have any difficulty implementing the functions in the `GAME` interface, you *should* change your representation—or at least think about it.

   You might be tempted to use mutable data to represent game state. **Don't!** The contract of the `GAME` interface requires that any value of type `config` be available to the AGS indefinitely. Mutating a configuration is not safe.

   If you think you might want *immutable* arrays, check out the `Vector` structure (see the ML supplement). (You can find out what's in any ML structure by typing, e.g., `open Vector` at the interactive prompt, or you can consult the Standard Basis documentation. You can also use Moscow ML's help system, e.g,

   ```
   - help "Vector";
   ```

   If you get interested in vectors, don't overlook the function `Vector.tabulate`.)

   One more thing. You may be tempted to start out by representing the contents of a square on the board using 0 and 1 or other arbitrary values. If you go this route, why not use `Player.player option`? It will make your program more elegant and easier to understand.
b. Choose a representation for moves. That is, write `move`. Everything said for configurations applies here also, but this choice seems less critical.
c. Declare the exception `Move`.
d. Write the function `initial`.
e. Write the function `whoseturn`.
f. Write `makemove`. The contract requires it to be Curried.
g. Write `outcome`. If the configuration is not final and nobody has won, return `NONE`.
   Hints for Tic-Tac-Toe:
   c. You could write a function which checks lines, another that checks columns and finally one that checks diagonals. Then `outcome` could call these functions with the right parameters.
   d. You could try pattern matching. Standard ML supports pattern matches on vectors by, e.g., `case a of #[x, y, z] => ....`

h. Write finished. This function should return true if somebody has won or if no move is possible (everybody is stuck). Be smart and use another function to do most of the work.
i. Write possmoves. This function must return a list of the possible moves (in no particular order). It is in everybody's interest that the list have no duplicates. *If the game is over, no further moves are possible*, and possmoves must return `nil`. (In this case, according to contract, finished must return `true`.)

   If you want to be clever, you can exploit rotation and reflection symmetries to prune the list returned by possmoves. You may be surprised how much difference this makes to performance. **For extra credit,**

   1. submit a version of possmoves that exploits symmetry to minimize the number of possible moves
   2. give a ``back of the envelope'' estimate of the time to be saved when the AGS plays against itself
   3. measure the actual time savings using the `Timer` and `Time` structures thusly:

   ```
   fun time f arg =
     let val start = Timer.startRealTimer()
         val answer = f arg
         val endit = Timer.checkRealTimer start
     in  print ("Time is " ^ Time.toString endit ^ "\n");
         answer
     end
   ```

   You can also try `startCPUTimer` and `checkCPUTimer`, but the answers you get are a bit more complicated.
j. Write `Move.toString`. This function must return a string of the form ``Player... moves to ...'' which does *not* end in a newline. You can build your strings using concatenation (`^`) and exported functions from other modules (e.g. `Player.toString`). To convert integer values to strings you can use the function `Int.toString`.

   Try to write `Move.toString` in such a way that `Move.fromString` and `Move.toString` cannot possibly be inconsistent, even if you make a mistake. (*Hint: how should you represent a bidirectional map between our names for locations and your internal representation of locations?*)
k. Write toString. You must return a simple ASCII representation of the state of the game configuration. The value should end in a newline. Don't forget to include the player whose turn it is to move. Give us more than a simple list of numbers. You can print a nice little ``ASCII graphics'' layout using only a few characters. To get you started, here is some untested sample code to print a row; it has type `player option list -> string`:

   <sample function rowString>=
   ```
   local
     fun boxString (SOME p) = Player.toString p
       | boxString (NONE  ) = " "
   in
     fun rowString []                 = "|\n"
       | rowString (box :: boxes) =  "| "   ^ boxString box ^ " " ^ rowString boxes
   end
   ```

   Defines boxString, rowString (links are to index).

   `Move.toString` and toString are not involved in the correctness of the AGS; they are used by the interactive player to show you what's happening. The better your output, the more fun it will be to play. You can see a simple sample by running `/comp/105/bin/ttt`.
l. Write `Move.prompt`. It takes the player whose turn it is to move, and it returns a prompt message (without newline) asking the specified player to give a move in the format we specified (naming the square).
m. Write `Move.fromString`. This function should take a string (which is probably the reply given after a call to `Move.prompt`, and it should return the move corresponding to that string. If there is no such move, it should raise an exception.

   You should try to write `Move.fromString` in such a way that `Move.fromString` and `Move.toString` cannot possibly be inconsistent, even if you make a mistake.

Be sure to try your functions on simple configurations.

Hints: You may find it useful to define a `structure Grid` *that you can use to represent a square or rectangular array of values of type* `'a`*. Defining suitable analogs of* `map` *and* `fold` *on the grid will help, as will functions to extract sub-grids (rows and columns). If you then define reflection and rotation on grids, you can easily do the extra credit.*

The most common mistake on this problem is to permit players to continue to move even when the game is over.

Bob Harper's code for Tic-Tac-Toe is 146 lines of Standard ML. I have a slicker version at only 87 lines&emdash;and it is four times faster. It works by exploiting bit-level parallelism using the `Word` structure and by flagrantly disregarding most of the hints given above.

Testing your code: useing the AGS with Tic-Tac-Toe

Exercise Abstract Game Solver (Difficulty *). --> To build a version of the AGS for ``Tic-Tac-Toe'' you must use the following command:

<example of creating a game-specific AGS>=
```
structure TTTAgs = AgsFun(structure Game = TTT)
```

Defines `TTTAgs` (links are to index).

Of course, I can't do any of this until I use the Moscow ML `load` function to get access to `AgsFun` and `TTT`. Here is an example:

<transcript from an actual session>= **[D->]**
```
: nr@labrador 7147 ; mosml
Moscow ML version 2.10-2 (Tufts University, February 2011)
Enter `quit();' to quit.
- load "ags";
> val it = () : unit
- load "ttt";
> val it = () : unit
- structure TTTAgs = AgsFun(structure Game = TTT);
> structure TTTAgs :
  {structure Game :
     {structure Move :
        {type move = move,
         exn Move : exn,
         val fromString : string -> move,
         val prompt : player -> string,
         val toString : player -> move -> string},
      type config = config,
      val finished : config -> bool,
      val initial : player -> config,
      val makemove : config -> move -> config,
      val outcome : config -> outcome option,
      val possmoves : config -> move list,
      val toString : config -> string,
      val whoseturn : config -> player},
   val bestmove : config -> move option,
   val forecast : config -> string}
-
```

This functor application creates a structure that implements the `AGS` signature:

<ags-sig.sml>=
```
signature AGS = sig
  structure Game : GAME
                          (* Given a configuration returns the
                           * most beneficial move for the player
```

```
                          * to move *)
  val bestmove : Game.config -> Game.Move.move option

                         (* Given a configuration computes the
                          * maximum benefit which can be
                          * obtained against an optimum
                          * player. The benefit is converted
                          * to a printable representation  *)
  val forecast : Game.config -> string
end
```

Defines AGS, bestmove, forecast, Game (links are to index).

The function `bestmove` returns the best move in a configuration, or `NONE` if no move is possible, i.e., the configuration is final. The function `forecast` returns a string predicting the outcome from a configuration if both players make perfect moves. The prediction, which should be `"Win"`, `"Loss"`, or `"Tie"`, is from the point of view of the player whose turn it is.

These functions can be ***slow*** because the AGS tries all possible combinations of moves. Be patient.

We have also provided you an interactive player. It uses the AGS so you must instantiate it to the Tic-Tac-Toe AGS using the following command:

<examples>= [D->]
```
structure P = PlayFun(structure Ags = TTTAgs);
```

Defines P (links are to index).

Again, to get `PlayFun` you will have to load the right module:

<transcript from an actual session>+= [<-D]
```
- load "play";
> val it = () : unit
- structure P = PlayFun(structure Ags = TTTAgs);
> structure P :
  {structure Game : ...
   exn Quit : exn,
   val getamove : player list -> config -> move,
   val play : (config -> move) -> config -> outcome}
-
```

The structure this application creates implements the following signature :

<play-sig.sml>=
```
signature PLAY = sig
  structure Game : GAME
  exception Quit
  val getamove : Player.player list -> Game.config -> Game.Move.move
    (* raises Quit if human player refuses to provide a move *)

  val play : (Game.config -> Game.Move.move) -> Game.config -> Player.outcome
end
```

Defines Game, getamove, PLAY, play, Quit (links are to index).

The function `getamove` expects a list of players for which the computer is supposed to play (the computer might play for X, for O, for both or for none). The return value is a function which the interactive player will use to request a move given a configuration. The idea is that the function returned will ask the AGS for a move if the computer is playing for the player to move, or will prompt the user and convert the user's response into a move.

The function `play` expects an input function (one built by `getamove`) and a starting configuration. This function then starts an interactive loop printing the intermediate configurations and prompting the users for moves (or asking the AGS where appropriate). One example is :

<examples>+= [<-D]
```
val computerxo = P.getamove [Player.X, Player.O]
       (*Computer plays for both X and O *)

val computero = P.getamove [Player.O]
       (*Computer plays only O *)

val cnfi = TTT.initial Player.X
      (* Empty configuration with X to start *)

val frustration = P.play computero
      (* We play against the computer *)

val _ = frustration cnfi
      (* A frustrating exercise *)
```

Defines `cnfi`, `computero`, `computerxo`, `frustration` (links are to index).

Playing Other Games

The code we supply includes a description of the game ``Nim''. The structure that implements ``Nim'' is called `structure Nim`. After you create an AGS solver and an interactive player for ``Nim'' you can play Nim with the AGS. The commands to instantiate AGS to ``Nim'' are:

<nim examples>=
```
structure NIMAgs = AgsFun(structure Game = Nim)
structure PN = PlayFun(structure Ags = NIMAgs)
```

Defines `NIMAgs`, `PN` (links are to index).

You play Nim by running `/comp/105/bin/nim`, but the user interface stinks.

We've also implemented a version of ``Connect 4'' that would be better called ``Connect 3'' (since 4 would be too slow). It is in `/comp/105/bin/four`.

Building an AGS

C.Implement an Abstract Game Solver (Difficulty *).** Given a configuration, an AGS should compute the *benefits* of all possible moves and pick the best one. More precisely, given a configuration and a player, the AGS assigns a benefit to that player of that configuration. A final configuration in which X has won should have maximum benefit to X and minimum benefit to O, and vice versa. Ties should have intermediate and equal benefit to both players. We compute the benefit of an intermediate configuration by looking at all possible moves and the benefits of the resulting configurations.

There are a variety of ways to view benefits; for example, we could assign larger benefits to winning quickly, and so on. For this assignment, however, it will be sufficient to consider three levels of benefits:

- Player to move can force a win
- Both players can force a tie
- Player to move can be forced to lose by his adversary

For **extra credit** you can prove that one of these three situations must hold in any game described by the `GAME` signature, provided that the game is deterministic and is guaranteed to terminate after finitely many moves.

Write an AGS using the following template:

<template for functor `AgsFun`>=

```
functor AgsFun (structure Game : GAME) : AGS = struct
  structure Game = Game

  fun bestresult conf = ...
  fun bestmove conf = ...
  fun forecast conf = ...
end
```

Defines AgsFun, bestmove, bestresult, forecast, Game (links are to index).

Note that the AgsFun definition uses the plain colon (:), not the opaque signature match :>. This means that the identity of the types Game.Move.move and Game.config is allowed to ``leak out.'' An alternative is to write the functor this way:

<alternative template for functor AgsFun>=

```
functor AgsFun (structure Game : GAME) :> AGS
   where type Game.Move.move = Game.Move.move
   and   type Game.config    = Game.config
= struct
  structure Game = Game

  fun bestresult conf = ...
  fun bestmove conf = ...
  fun forecast conf = ...
end
```

Defines AgsFun, bestmove, bestresult, forecast, Game (links are to index).

Whichever way you write the functor, the function bestresult is a suggested helper function that should return a pair of values:

- The best move, if any, for the player to take in the current configuration. This should be a value of type Game.Move.move option, so if the configuration is final, you can return NONE.
- The predicted outcome of the game if both players play perfectly. It suffices to use an outcome of type Player.outcome, but you can play around with this one some&emdash;for example, you might want to return an outcome like ``Player X wins in 3 moves.'' This would help you build an aggressive AGS.

  You might be tempted to use a ``relative'' outcome like ``Win, Lose, or Tie.'' This can be made to work, but it is harder to get right, especially in games where players don't always take turns.

In order to make bestresult work, you'll need some recursive calls. You'll also want a helper function that lets you compare the benefits of different outcomes, so bestresult can choose the most desirable outcome for the current player.

Hints:

- To speed up the AGS, you may want to stop the search as soon as you find a forced win.
- Do **not** assume that players take turns, that the last player to move always wins, or any other properties of Tic-Tac-Toe. Use whoseturn and outcome instead. We will test your AGS on games that are quite different from Tic-Tac-Toe.

To test your AGS, you'll need to replace our ags.ui and ags.uo files with the ones you compile from your source code. At this point you'll be able to run the same test cases you used earlier, as well as what's in part B.

My AGS takes 34 lines of Standard ML.

One common mistake to avoid

If you build a simple AGS that fits in less than 40 lines of code, it is not going to try to fool you: if you can force a win, the AGS will pick a move more or less arbitrarily. A simple AGS has no notion of ``better'' or ``worse'' moves; it knows only whether it can force a win.

Here's the common mistake: you're playing against the AGS, and it makes a terrible move. You think it's broken. For example, suppose you are playing X, the AGS is playing O, and you start play in this position:

```
-------------
|   | O |   |
-------------
|   | X |   |
-------------
|   |   |   |
-------------
```

You move in the upper left corner. **The AGS does not move lower right to block you.** Is it broken? No—the AGS recognizes that you can force a win, and it just gives up.

If you want an AGS that won't give up, you need to implement an aggressive version that will delay the inevitable as long as possible. An aggressive AGS searches more states so that it can (a) win as quickly as possible and (b) hold on in a lost position as long as possible.

My aggressive AGS is under 60 lines of Standard ML code.

Descriptions of the games

Here are descriptions of 3 games: ``Tic-Tac-Toe'', ``Nim'' ``and ``Connect 4''. Do not worry if you haven't seen the game before&emdash;you can learn by playing against a perfect or near-perfect player. (The Connect 4 player would be perfect if it were faster.) For the purpose of this assignment you do not have to know any tricks of the games but only to understand their rules.

Tic Tac Toe

This is an adversary game played by two persons using a 3x3 square board. The players (traditionally called X and O) take turns in placing X's or O's in the empty squares on the board (player X places only X's and O only O's). The board is empty in the initial configuration.

The first player who managed to obtain a full line, column or diagonal marked with his name is the winner. The game can also end in a tie. In the picture below the first configuration is a win for O, the next two are wins for X and the last one is a tie.

```
-------------    -------------    -------------    -------------
| X |   | X |    |   |   | X |    | X | O |   |    | O | O | X |
-------------    -------------    -------------    -------------
|   | X |   |    | O | X | O |    | X | O |   |    | X | X | O |
-------------    -------------    -------------    -------------
| O | O | O |    | X |   | O |    | X |   | O |    | O | X | O |
-------------    -------------    -------------    -------------
```

In this game a player who plays perfectly cannot lose. All your base are belong to the AGS.

Nim

This is an adversary game played by two persons. The game is played with number of sticks arranged in 3 rows. In the initial state the rows usually contain 3, 5 and 7 sticks respectively. The players take turns in removing sticks: each player can remove 1, 2 or 3 adjacent sticks from one row. The one that removes the last stick is the loser. Or, stated differently the first player who has no sticks to remove is the winner. Below were presented two configurations. The first one is the initial configuration (for the 3, 5 and 7) case and the other one is the configuration obtained after a few moves. A possible sequence of moves that might lead to this configuration is:

1. X removes sticks 0, 1 and 2 from row 1

2. O removes stick 1 from row 0
3. X removes stick 6 from row 2
4. O removes sticks 3 and 4 from row 2

```
Row 0: | | |                  | _ |

Row 1: | | | | |              _ _ _ | |

Row 2: | | | | | | |          | | | _ _ | _
```

We have represented a stick using a ``|''and a missing stick using a ``_''. It might be wise to play with a smaller configuration (2, 3 and 4 for example) because otherwise the AGS will take too long to produce its answers.

For this game the first player can always win no matter what the other does. If you let the AGS start you have no chance. If you play first you can beat the AGS, but you have to play well.

Connect 4

This is an adversary game played by two persons using 6 rods and 36 balls. Imagine the rods standing vertically, and each ball has a hole in it, so you can drop a ball onto a rod. The balls are divided in two equal groups marked X and O. The players take turns in making moves. A move for a player consists in sliding one of its own balls down a rod which is not full (the capacity of a rod is 6). The purpose is to obtain 4 balls of the same type adjacent on a horizontal, vertical or diagonal line. The game ends in a tie when all the rods are full and no player has won. We represent below the initial configuration of the game and a final state where X has won.

```
| | | | | |             | | | | | |
| | | | | |             | | | | | |
| | | | | |             | | | | | |
| | | | | |             O | | | | |
| | | | | |             O | O | | |
| | | | | |             O X X X X |
-----------             -----------
```

Our version uses 5 rods and connects 3, because otherwise the AGS takes too long.

Extra Credit

Symmetry. Speed up Tic-Tac-Toe by exploiting symmetry as described above.

Proof. Prove the ``forcing'' property of these simple games as described above above.

Four. Implement Connect 4.

Game. Suggest another simple adversary game, and (with the instructor's approval) implement it. The game should be small with a small number of possible moves; otherwise the exhaustive search is infeasible.

Aggression. With the simple benefits outlined above, the AGS will ``give up'' if it can't beat a perfect player---all moves are equally bad, and it apparently moves at random. What this scheme doesn't account for is that the other player might not be perfect, so there is a reason to prefer the most distant loss. In the dual situation, when the AGS knows it can win no matter what, it will pick a winning move at random instead of winning as quickly as possible. **This behavior may lead you to suspect bugs in your AGS. Don't be fooled**.

Change your benefits so that the AGS prefers the closest win and the most distant loss. (This means you can only prune the search if you find a win in one move.) If you are clever, you can encode all this information in one value of type `real`.

Learning. We can re-use the `GAME` signature for more than one purpose. Implement a ``matchbox'' learning engine in the style explained by Martin Gardner's article on the reading list. You can use the SML/NJ library to store state with each configuration, using the following signautre:

```
signature ORDERED_GAME = sig
  include ORD_KEY
  include GAME
  sharing type conf = ord_key
end;
```

You may have to modify the AGS to notify each player of the outcome of the game. See me for more help with details.

What code we give you and how to compile

In the git repository at `/comp/105/git/ttt`, you'll find sources for most of the signatures, structures, and functors in this assignment. You'll also find an AGS in binary form only. And you'll find a `compile` script, which should compile your code using the Moscow ML compiler, `mosmlc`.

<example transcript of compilation>=
```
mosmlc -c -toplevel game-sig.ui player.ui ttt.sml
```

This compilation produces two files:

- `ttt.ui`, which can be used on the command line when compiling other units that depend on TTT.
- `ttt.uo`, which contains the compiled binary

You can do two things with the `.uo` files:

- Load them directly into an interactive session, e.g.,

  ```
  : nr@labrador 2856 ; mosml
  Moscow ML version 2.10-2 (Tufts University, February 2011)
  Enter `quit();' to quit.
  - load "ttt";
  > val it = () : unit
  - open TTT;
  > structure Move :
    {type move = move,
     exn Move = Move : exn,
     ...}
    type config = config
      ...
    val finished = fn : config -> bool
      ...
  ```

  Once you load a module, you cannot recompile it and reload it later. You have to start Moscow ML over again, or perhaps you can recompile from within the interactive session (I'm not sure about this one).
- You can link a bunch of `.uo` files together to form an executable binary. To do anything interesting, one of the source files should have a top-level call to `play`, `forecast`, or some other interesting function.

What to submit

For this assignment you should use the script `submit105-sml` to submit

- Either `README`
- For problem 1, file `arbitrary.sml`
- For problem 2, files `heap-sig.sml` and `heapsort.sml`
- For part A, file `ttt.sml`
- For part C, file `ags.sml`

The ML files should contain all structure and function definitions that *you* write for this assignment (including any helper functions that may be necessary), in the order they should be compiled. The files you submit must compile with Moscow ML, using the compile script we give you. We will reject files with syntax or type errors. Your files should compile *without warning messages*. If you must, you can include multiple structures in your files, but *please don't make copies of the structures and signatures above*; we already have them.

Acknowledgments

This assignment is derived from one graciously provided by Bob Harper. George Necula, who was his teaching assistant at the time (and is now a professor at Berkeley and is world famous as the inventor of proof-carrying code), did the bulk of the work.

Compiling Standard ML using MLton

If you find that your games are running too slow, you may want to try compiling them with MLton. MLton is a whole-program compiler; all your modules must be in a single file. For example:

```
cat player-sig.sml player.sml game-sig.sml ags-sig.sml play-sig.sml \
  play.sml ttt.sml ags.sml mrun.sml > playttt.sml
mlton -output ttt -verbose 1 playttt.sml
```

Because MLton requires source code, you will be able to use it only once you have your own AGS. More information about MLton is available on the man page and at `www.mlton.org`.

Cross-reference

ML Identifiers

Code chunks