CS152 Homework: Standard ML Modules

Due Wednesday, May 1 at 12:10AM (aka Tuesday night at midnight)

The purpose of this assignment is threefold:

- 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.
- To get a preview of exhaustive search in a less general, more efficient context than Prolog

---

Playing Adversary Games

You will implement a system for playing simple adversary games. The program will show game configurations, accept moves from the user and choose the best move. Additionally, we will have a function that will help us find out if the program is unbeatable.

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. As the procedure is based on exhaustive searching (trying all valid moves and selecting the best), it is general enough that we can abstract it. If the solving procedure were highly dependent on the game, such a separation would not be easy to make.

An important issue in such a modular design is to determine the interface between the AGS and the particular game. Such an interface is specified using the SML signature `GAME`. This signature declares all the data and functions that an AGS must know about the game in order to do its job. We must be careful to make the signature general enough to cover a wide variety of games. Even details like ‘‘the players take turns’’ are considered to be part of the game and not the AGS and will therefore be hidden in the game abstraction. You could even implement a solitaire as a ‘‘two-player’’ game in which the second player never gets a turn!

The idea behind the AGS is that it tries all possible moves. For each move, it takes the resulting configuration and tries all possible moves for the adversary, and so on. It assumes at each level that the player plays perfectly, that is, he plays the move most likely to result in 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. You may have seen the ‘‘alpha-beta’’ pruning strategy in CS 51. We won’t worry about these variations; we will just implement the basic algorithm.

The assignment has three parts. You will implement a particular game, exercise the AGS on that game, then finally implement an AGS of your own.

Players and Winners

We start by describing a few basic functions which are independent of the game but related to the fact that there are two adversaries. The specifications for these functions are given by the signature `PLAYER` and their implementation is given to you in a structure `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

                             (* Pretty print. Convert a value to a
                              * printable representation  *)
  val toString : player -> string
  val outcomeToString : outcome -> string
end
```

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

Here's the implementation:

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

  fun otherplayer X = O
    | otherplayer O = X

                            (* Returns the name of a player *)
  fun toString X = "X"
    | toString O = "O"
                            (* The name of the winner *)
  fun outcomeToString TIE = "Tie"
    | outcomeToString (WINS p) = toString p
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. Whenever you refer to a player in your code you will have to use one of the following notations: `Player.otherplayer p` or `Player.X` or `Player.O` or `Player.WINS p`. Note that the last three expressions can also be used as patterns.

Specification of a Game

Next we describe the specification of a game (the information our AGS needs to know about the game in order to be able to play it ‘‘intelligently’’). Such a description must match the signature `GAME`. Read the comments for each type and each value in the signature.

```
<game-sig.sml>=
```

```
signature GAME = sig
  structure Move : sig  (* information related to moves *)
    eqtype move             (* A move (probably 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 ..."  *)
  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 *)

  val toString : config -> string
                  (* Returns an ASCII representation of the
                     configuration.  It 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 because
                     occasionally the solver needs to know whose
                     turn is and it cannot extract this
                     information from the configuration by itself
                     because it doesn’t know the layout 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
                  (* Returns the outcome of the game, or if the game isn’t over
                     (the configuration is not a final one), returns 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 *)
```

```
end
```

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

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`.

When you are debugging, it is tremendously useful to get these 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. **Remember that 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. [50 points] Implement the description for ‘‘Tic-Tac-Toe.’’ For a verbal description of the game refer to the end of this assignment. Call your description 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).

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. No other code will be able to use them outside---they will not be exported, because we've forced structure `TTT` to have signature `GAME`.

So we can test your code, we insist that you use the following names of squares in `showmove` and `getmove`:

```
 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 returned by `getmove`.

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 is a crucial step because most of the complexity of the implementation depends on this choice. Choose a representation which will make your life easy when it comes to implementing `makemove`, `possmoves`, and `outcome`. You must understand all the procedures that you will implement on your data structure *before* you fix a representation. There are many possible representations; any one is OK provided you are able to implement the functions required by the signature. The AGS, which is the client of your structure, doesn't care how you implement the specification as long as you do it.

   You might be tempted to use mutable data to represent game state. *Don't even think of it!* The problem is that you can't tell when it is safe to mutate the state, because you never know when the AGS might go back to it. 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.)

   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 seems to be a less critical choice.

c. Declare the exception `Move`.

d. Write the function `initial`.

e. Write the function `whoseturn`.

f. Write `makemove`. Pay attention to the specified type. There are other possible types but your clients (AGS in this case) assume that the type is as specified.

g. Write `outcome`. In the case of ‘‘Tic-Tac-Toe’’ you might write a function which checks lines, another that checks columns and finally one that checks diagonals. Then you call these functions with the right parameters. (This is only a suggestion and you might find better ways of doing it, especially if you come up with a righteous pattern match or other sneaky tricks. Standard ML supports pattern matches on vectors by, e.g., `case a of #[x, y, z] => ...`) If the configuration is not final and nobody has won, then return a tie.

h. Write `finished`. This function should return true if somebody has won or if no move is possible (everybody is stuck). Be smart and see if you can use some other functions to do part of the job.

i. Write `poss_moves`. This function must return a list of the possible moves (in no particular order). It is in everybody’s interest that the list have only unique elements. You can achieve this implicitly by having a smart algorithm to collect the possible moves.

   One thing that AGS assumes that is *not* captured in the signature is that if the function `finished` returns false on a configuration then `poss_moves` will return a non-empty list on that configuration.

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

   1. submit a version of `poss_moves` 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 `showmove`. 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.pp_player`). To convert integer values to strings you can use the function `Int.toString`.

You should try to write `showmove` in such a way that `getmove` and `showmove` 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 `showconf`. 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.

   `showmove` and `showconf` 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 against the AGS. You can see a simple sample by running `~cs152/bin/ttt`

l. Write `getmove`. This one's a bit tricky. It takes the player whose turn it is to move, and it returns a pair. The first element of the pair is a prompt message (without newline) asking the specified player to introduce its move in the format we specified (naming the square). The second element is a function used to convert the reply into a move in your representation (actually a `move option`). Be sure to try your functions on simple configurations.

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

Hint: 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.

Bob Harper's code for Tic-Tac-Toe is 146 lines of Standard ML. I have a slicker version at only 87 lines---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.

Use AGS with Tic-Tac-Toe

B. [10 points] Exercise the AGS system. To build a version of the AGS for ''Tic-Tac-Toe'' you must use the following command:

```
<examples>= [D->]
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 2.00 (June 2000)
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 cnfi = TTT.initial Player.X
      (* Empty configuration with X to start *)

val itself = P.play computerxo ;
      (* The computer playing by itself *)

itself cnfi
      (* Lets the computer play alone starting in the initial config *)
```

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

This exercise has the following parts:

1. Write the function `compx:TTT.config->Player.winner` which will simulate the computer playing for X from any starting configuration. (This function should have a behavior similar to `itself` in the example above). Although your function accepts a `TTT.config` as argument, it should not depend in any way on the details of `TTT`, but only on the fact that `TTT` matches signature `GAME`.

2. Compute the forecast of the following configuration (O is to move) (use the function `TTTAgs.forecast`):

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

3. using `TTTAgs.forecast` show that AGS cannot lose a game starting in the initial configuration (no matter who plays first). My code takes a few milliseconds to do this, but it's OK if yours takes significantly longer.

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 `~cs152/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 `~cs152/bin/four`

Building an AGS

C. [40 points] Implement an Abstract Game Solver. We can summarize the operation of the AGS by saying that in a particular configuration it evaluates the *benefits* of all possible moves and picks the best one. We could phrase this more precisely as follows: given a configuration and a player, assign a benefit to that player of that configuration. A final configuration in which X has one 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 only that the game is guaranteed to terminate after finitely many moves.

Write an AGS using the following template:

```
<template for functor Ags>=
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).

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 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---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.

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---you can learn by playing against a quasi-perfect player (it would have been 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. I doubt one can beat 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 4from 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 (also on reserve in McKay). 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.

Cross-reference

ML Identifiers

Code chunks

What code we give you and how to compile

In directory `~cs152/homework/sml`, 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. You'll 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 2.00 (June 2000)
  Enter `quit();' to quit.
  - load "ttt";
  > val it = () : unit
  - open TTT;
  > type config = ...
    type move = ...
    exn Move = Move : exn
    val finished = fn : config -> bool
    val getmove = fn : player -> string * (string -> move option)
      ...
  ```

  Note that **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. This binary has to have some initialization code to do anything interesting.

What to submit

For this assignment you should submit the files `ttt.sml` for part A, `exercise.sml` for part B, `ags.sml` for part C, and either `README` or `README.ps`. 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 Makefile we hope to give you; we will reject files with syntax or type errors. Your files should compile *without warning messages*; we will deduct points for compiler warnings. Your `exercise.sml` file should be usable with `use "exercise.sml";`. 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, his TF at the time (now a professor at Berkeley), did the bulk of the work.

Running Standard ML of New Jersey

If you find that Moscow ML is too slow for you, you may want to try the Standard ML of New Jersey compiler. This is an interactive compiler; the binary is `/usr/local/sml/bin/sml-cm`. For separate compilation, it uses the Compilation Manager, which is like a ''smart'' version of `make`; you name your sources and it compiles them. When a source change, the compilation manager recompiles the mininum necessary to account for the change.

Instead of a `Makefile`, CM uses the file `sources.cm`. We can provide binary versions of everything except `TTT`, so your `sources.cm` will look like this:

```
<sources.cm for part A>=
group is
  /home/c/s/cs152/lib/sml/ttt/ags/sources.cm
  ttt.sml
```

The first line tells CM to get all the AGS and other support code; the second line names your source file. ***Don't* copy the structures and signatures from this assignment!** They are already provided in `/home/c/s/cs152/lib`, and if you make copies, your code will break (because of generativity).

To use the compilation manager, run `sml-cm` and type

```
CM.make();
```

at the ML prompt. If your program compiles successfully, CM will load all the signature, structure, and functor definitions into the ML top level, and you can try the examples from *<examples>*. Otherwise you will have to correct your errors and try again.

To test your AGS with Standard ML of New Jersey, you'll need to use your version of Tic-Tac-Toe, plus the following `sources.cm` file:

```
<sources.cm for supplying your own AGS>=
group is
  /home/c/s/cs152/lib/sml/ttt/sources.cm
  ags.sml
  ttt.sml
```

Note: If you use emacs, you will find there is a very useful sml-mode for use with Standard ML. The C-c ‘ command is especially valuable, as it will jump straight to an error if found.