Modules and Abstract Types

CS 105 Assignment

Due Tuesday, November 22, 2022 at 11:59PM

Contents

Overview

To build systems at scale, we break them into modules, and we put abstraction barriers between the modules. The most effective abstraction barrier we have is *data abstraction*. It's a key element in the design of systems implemented in Ada, C, C++, Eiffel, Go, Haskell, Java, and related languages. Modules and abstract types are key mechanisms in Standard ML, which, not coincidentally, has one of the most expressive and powerful system-level abstraction mechanisms ever created.

In this assignment, you will

- Practice using interfaces as they are found in ML, Java, and Go
- Learn about exploratory design by writing client code for an interface that is not yet implemented
- Write a client for an interface that has multiple implementations
- Revisit arbitrary-precision arithmetic, which you will see yet again in the last homework
- Learn something about two-player games of complete information

You'll do this in two parts:

- In part one, you use an *unknown* implementation of natural numbers to implement *signed* integers (exercise I).
- In part two, you implement an unsophisticated (yet unbeatable) computer opponent that can play two-player games. You will also implement a two-player game. The computer opponent requires careful attention to the design process but not a lot of code: in essence, it boils down to one carefully crafted recursive function. The game will be your choice among three, most of which require less thought and more code than the opponent.

You may do this work on your own or with a partner.

> Note: Because you're creating modules, not just functions, and you're also learning some new technology, you'll do a lot of reading before you write your first line of code. This experience makes the assignment feel time-consuming, but overall, past students have reported spending the same amount of time on this assignment as on other assignments.

Setup

The code in this handout is installed in `/comp/105/lib`. Your own code will be compiled using a special script, `compile105-sml`, which is available on the servers via `use comp105`. This script does not produce any executable binaries. Instead, it creates binary modules ("`.uo` files") that you can load into Moscow ML, as in `load "bignum";`. The script can compile all files or just a single file:

```
compile105-sml
compile105-sml bignum.sml
```

Loading binaries into Moscow ML requires an additional argument to `mosmlc` and `mosml`, as in

```
mosml -I /comp/105/lib -P full
```

If anything surprises you, consult Appendix I below, which explains your options for compiling.

Dire warnings

Unless otherwise noted below, functions and constructs that were forbidden on earlier assignments remain forbidden.

- Functions `length`, `hd`, and `tl` are still forbidden.
- The syntactic form `open` is still forbidden.
- Auxiliary functions at top level are still forbidden—but inside a module, you may define as many auxiliary functions as you like.

In addition,

- ML's hashtag syntax for record fields is forbidden. (It is non-idiomatic, and it doesn't really have a type.) **To earn a passing grade, your code must not use `#1`, `#2`, `#recommendation`, `#expectedOutcome`, or any other form of this syntax.** Use pattern matching instead.
- Code must not contain **bracket faults**. Every file should pass `sml-lint` without errors or warnings. Run `sml-lint *.sml` early and often. (The `sml-lint` program is also run by the submit scripts, but if you wait until submission time, you'll regret it.)[1]

Reading

For this homework, you will need to understand the ML Modules section of "Learning Standard ML." The book chapter on modules, chapter 9, is not included in your abridged edition—the language there is a little too different from Standard ML. Instead of the book chapter, we recommend the sixth lesson on program design: "Program design with abstract data types."

Reading comprehension

Before starting the programming problems, answer the reading-comprehension questions for this module, which you will find online. It is also OK to alternate between reading-comprehension questions and related homework questions.

[1] If you copy a signature, `sml-lint` will complain. Don't copy signatures.

Arbitrary-precision signed integers

Standard ML's primitive type `int` is limited to machine precision. If arithmetic on `int` results in a value that is too large or too small to fit in one word, the primitive functions raise `Overflow`. In this part of the homework, you will implement a true integer abstraction, called `bigint`, which *never* raises `Overflow`. (The worst it can do is make your program run out of memory.)

Large integers are described by this interface:

```
signature BIGNUM = sig
   type bigint
   exception BadDivision               (* contract violation for sdivmod *)

   val ofInt   : int -> bigint
   val negated : bigint -> bigint      (* "unary minus" *)
   val <+>     : bigint * bigint -> bigint
   val <->     : bigint * bigint -> bigint
   val <*>     : bigint * bigint -> bigint
   val sdivmod : bigint * int -> { quotient : bigint, remainder : int }

     (* Contract for "short division" sdivmod, which is defined only on
        *nonnegative* integers:

        Provided 0 < d <= K and n >= 0,
          sdivmod (n, d) returns { quotient = q, remainder = r }
        such that
          n == q /*/ ofInt d /+/ ofInt r
          0 <= r < d

        Given a negative n or d <= 0, sdivmod (n, d) raises BadDivision.

        Given d > K, the program halts with a checked run-time error.
        The constant K depends on the number of bits in a machine integer,
        so it is not specified, but it is known to be at least 10.
     *)

   val compare : bigint * bigint -> order

   val toString : bigint -> string

     (* toString n returns a string giving the natural
        representation of n in the decimal system.  If n is
        negative, toString should use the conventional minus sign "-".

        And when toString returns a string containing two or more digits,
        the first digit must not be zero.
     *)

  val invariant : bigint -> bool  (* representation invariant---instructions below *)
```

```
end
```

You will *not* build your implementation from scratch. Instead, you will use ML's functor mechanism to build on top of natural numbers. Natural numbers implement this interface:

```
signature NATURAL = sig
   type nat
   exception Negative     (* the result of an operation is negative *)
   exception BadDivisor   (* divisor zero or negative *)

   val ofInt   : int -> nat          (* could raise Negative *)
   val /+/     : nat * nat -> nat
   val /-/     : nat * nat -> nat    (* could raise Negative *)
   val /*/     : nat * nat -> nat
   val sdivmod : nat * int -> { quotient : nat, remainder : int }

     (* Contract for "Short division" sdivmod:

        Provided 0 < d <= K,
          sdivmod (n, d) returns { quotient = q, remainder = r }
        such that
          n == q /*/ ofInt d /+/ ofInt r
          0 <= r < d

        Given a d <= 0, sdivmod (n, d) raises BadDivisor.

        Given d > K, the program halts with a checked run-time error.
        The constant K depends on the number of bits in a machine integer,
        so it is not specified, but it is known to be at least 10.
     *)

   val compare : nat * nat -> order

   val decimal : nat -> int list

     (* decimal n returns a list giving the natural decimal
        representation of n, most significant digit first.
        For example,  decimal (ofInt 123) = [1, 2, 3]
                      decimal (ofInt 0)   = [0]
        It must never return an empty list.
        And when it returns a list of two or more digits,
        the first digit must not be zero.
     *)

  val invariant : nat -> bool  (* representation invariant---instructions below *)

end
```

In this homework, you implement `BIGNUM` as a functor that takes an implementation of `NATURAL` as a parameter (Exercise I).

Exercise I: Integers from natural numbers (30%)

Abstract data type: large integers. In file `bignum.sml`, define a functor `BignumFn` that takes as its argument a structure `N` matching signature `NATURAL`, and returns as its result a structure matching signature `BIGNUM`. Your functor should look like this:

```
functor BignumFn(structure N : NATURAL) :> BIGNUM
  =
struct
  ... sequence of definitions here ...
end
```

Within the body of the functor, you refer to the representation of natural numbers as type `N.nat`, and you call operations using the fully qualified names of the functions as in `N./+/ (n, m)`.

ML syntax hint: The introduction form for the quotient/remainder record is

`{ quotient =` e_1`, remainder =` e_2 `}`

for any expressions e_1 and e_2.

The elimination form is

`let val { quotient =` q`, remainder =` r `} = …`

for any *patterns* q and r. (In typical usage, q and r will both be names.) You can see some examples in the section on implementing short division in "Mastering Multiprecision Arithmetic".

How big is it? My implementation is about 70 lines, of which about 15 are blank.

Representation, abstraction function, and invariant

What you need to know about an integer is how big it is and whether it is negative: its *magnitude* and *sign*. Within that constraint, you have your choice of representations. Several representations will work; here are three good ones:

- Represent the magnitude and sign independently.
- Encode the sign in a value constructor, and apply the value constructor to the magnitude, as in `NEGATIVE mag`.
- Define *three* value constructors: one each for positive numbers, negative numbers, and zero. A value constructor for a positive or negative number is applied to a magnitude. The value constructor for zero is an integer all by itself.

Each of these representations has its advantages, and they all work. Pick what you think will make your job easy.

Document your representation by writing down the abstraction function and invariant:

- Define the abstraction function *in a comment*. We recommend defining the abstraction function using algebraic laws. The right-hand sides should use standard arithmetic notation for operations on the mathematical integers.

- Write the invariant in an ML function named `invariant` of type `bigint -> bool`, located *inside* the module. Because it is located inside the module, the `invariant` function has complete access to the representation of `bigint`.

  The `invariant` function must typecheck. You may unit-test it if you wish, but you don't have to.

  It is OK if the `invariant` function for `bigint` is not interesting—depending on your chosen representation, it might even be a function that returns `true` on every input.

Put the abstraction function and invariant *inside* your `BignumFn` functor, right after the definition of type `bigint`.

Two mild warnings

This abstraction has just one pitfall: the integer zero is likely to have two or even three different representations. **You must make it impossible for client code to distinguish them.** That is, it must be impossible for anyone to write a program that *uses* the `BIGNUM` interface and can tell one representation of zero from another. The risks are greatest in exported functions `compare` and `toString`.

There is also a minor pitfall: if an adversary hands you the most negative integer (see `Int.minInt`), its magnitude cannot be represented as a machine integer. For example, if a machine integer had only four bits, it would represent integers in the range from -8 to 7, and if someone were to ask you to convert -8, you wouldn't be able to represent *positive* 8 as a machine integer.

To compute the magnitude of a negative integer safely, use the following steps:

1. Add one to the negative integer (from -8, this would give -7)
2. Negate the sum (this would give 7, no worries!)
3. Convert the result to `nat`
4. Add one to the `nat` (no worries; `nat` can be arbitrarily large!)

You would be wise to write a test case involving `Int.minInt`.

Recommendation: infix operators

Inside your `BignumFn` functor, I recommend you change the "fixity" of the major operators so you can write both calls and definitions using infix notation. You can change the fixity using these declarations:

```
infix 6 <+> <->
infix 7 <*> sdivmod
```

If you also want to use the operations from the `NATURAL` interface as infix operators, try something like this:

```
val /+/ = N./+/
val /-/ = N./-/
val /*/ = N./*/

infix 6 /+/ /-/
infix 7 /*/
```

Following these declarations, you can write both definitions and calls using infix notation:

```
fun thing1 <+> thing2 = ...
```

```
... mag1 /+/ mag2 ...
```

Guidance: algebraic laws

Arithmetic on signed integers requires algorithms you may not have thought about since elementary school. The heavy lifting is done in the implementation of natural numbers (below); the integer abstraction mostly has to get the signs right. To help you get signs right, we provide some algebraic laws. In the laws, magnitudes appear as variables `N` and `M`; signs appear as symbols `+` and `-`, and the `BIGNUM` and `NATURAL` operations are written using infix notation.

- Multiplication:

  ```
  +N <*> +M  ==  +(N /*/ M)
  +N <*> -M  ==  -(N /*/ M)
  -N <*> +M  ==  -(N /*/ M)
  -N <*> -M  ==  +(N /*/ M)
  ```

- Addition:

  ```
  +N <+> +M  ==  +(N /+/ M)
  +N <+> -M  ==  +(N /-/ M)  ==  -(M /-/ N)
  -N <+> +M  ==  -(N /-/ M)  ==  +(M /-/ N)
  -N <+> -M  ==  -(N /+/ M)
  ```

  Warning: As shown in the laws, adding integers of opposite signs requires subtracting magnitudes. Unless the magnitudes are equal, only one of the subtractions will work—the other will raise exception `N.Negative`.

- Subtraction can be implemented by changing the sign of the subtrahend and adding the result to the minuend:[2]

  ```
  +N <-> +M  == +N <+> -M
  ... and so on ...
  ```

- Short division is defined only on nonnegative integers, and it just delegates to `N.sdivmod`.

- When "signed zeroes" are involved, comparison becomes tricky. A call to `compare` mustn't be able to distinguish a "plus zero" from a "minus zero." That is, the following algebraic law must hold:

  ```
  compare (+0, -0)  ==  EQUAL
  ```

  Here are some more general laws:

  ```
  compare (+N, +M) = N.compare (N, M)
  compare (-N, -M) = N.compare (M, N)   (* order is swapped *)
  compare (-N, +M) = LESS,    provided N is not 0 or M is not 0
  compare (+N, -M) = GREATER, provided N is not 0 or M is not 0
  ```

Guidance: short division

If it gets a divisor that's too big, function `sdivmod` is allowed to fail with a checked run-time error. How big is too big? Anything that makes division on *natural numbers* fail. In other words, your `BignumFn`

[2] Words like "subtrahend" and "minuend" are useful, but I always have to look them up.

only has to check the sign of the dividend and divisor; it doesn't have to worry about the magnitude of the divisor.

Related reading: Using the information in the ML learning guide, read about ML signatures, structures, and functors.

Testing your arithmetic

Signed integers are much less tricky than natural numbers, and they don't require extensive testing. ~~Make sure every line of code you write is exercised by some test, and you'll be good to go.~~

Testing is optional, but if you're keen to do it, I provide some computer-generated tests for signed integers. These tests can be compiled using `compile105-sml`, but they're useful only if you also have an implementation of natural numbers. Such an implementation can be thrown together based on the model solutions to the ML homework—as long as you can implement `decimal`, you can just skip `sdivmod`. **If you create an implementation of `NATURAL` that uses the representation from the ML homework, you are welcome to share it with other students.**

Once you have the modules you need, you can load them into Moscow ML and run the tests:

```
- load "natural";
- load "bignum";
- load "bignum-tests";
- structure B  = BignumFn(structure N = Natural);
- structure BT = UnitTestsFun(structure Bignum = B);
- BT.run();
```

Avoid a common mistake

It's a surprisingly common mistake to write a `BIGNUM` implementation of `ofInt` that doesn't work with negative inputs.

It's a common mistake to think you need to write code in `BignumFn` to deal with `K`. The `K` in the division contract is dealt with entirely in the implementation of natural numbers. All you need to do is catch the exception from `NATURAL` and raise the exception from `BIGNUM`.

Two-player games

Data abstraction supports reuse. In this part of the assignment, you'll reuse code for an algorithm that can play a game. Because the game is abstract, your algorithm will be capable of playing any finite, two-player game of complete information. It will be tested on several!

- The game abstraction is defined by an interface (the ML signature `GAME`). This interface mixes abstract data types with "manifest" data types (exposed representations).[3]
- You'll implement the `GAME` interface. You'll pick one of the games on the games page, and you'll choose the representations of the game's key abstractions.
- You will reuse my code to play your game against a computer opponent.

[3]The abstraction is based on a design by George Necula.

- You will build your own computer opponent, the Abstract Game Solver, which will able to play any `GAME`.

An abstraction of two-player games

The data in a two-player game are as follows:

- A *player* may be either `X` or `O`.
- An *outcome* is either "one player wins" or "the game ends in a tie".
- A *state* is abstract.
- A *move* is abstract.

Every game is a state machine: the game starts in some *initial* state, and the game is played by transitioning from state to state. Each transition is triggered by a *move*. When the game reaches a *final* state, the game is over. The `GAME` interface enables software not only to drive the state transitions, but also to ask such questions as "what moves are legal in this state?", "is the game over?", and "whose turn is it?"

Manifest (exposed) types: Players and outcomes

The representations of players and outcomes are *exposed*. (In ML, exposed types are called "manifest". In other languages, they are "public.") The types are exposed by the signature `PLAYER`:

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

  val otherplayer : player -> player
  val unparse     : player -> string
  val outcomeString : outcome -> string
end
```

The signature `PLAYER` also includes some functions that compute with players and outcomes. Signature `PLAYER` is implemented by a structure called `Player`:

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

  fun otherplayer X = O
    | otherplayer O = X

  fun unparse X = "X"
    | unparse O = "O"

  fun outcomeString TIE = "a tie"
    | outcomeString (WINS p) = unparse p ^ " wins"
end
```

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.

First abstraction and abstract type: Moves

The `move` abstraction helps communicate gameplay to a human player. Functions `visualize`, `prompt`, and `parse` can respectively show, request, and understand moves.

```
signature MOVE = sig
  type move               (* A move---perhaps an (x,y) location *)
  exception Move          (* Raised for invalid moves *)

  (* CREATOR *)
  val parse : string -> move
        (* Converts the given string to a move; If the string
           doesn't correspond to a valid move, parse raises Move *)

  (* OBSERVERS *)
  val prompt : Player.player -> string
        (* A request for a move from the given player *)
        (* Example: "What square does player O choose?" *)

  val visualize : Player.player -> move -> string
        (* A short string describing a move.
           Example: "Player X picks up ...".
           The string may not contain a newline. *)
end
```

Second abstraction: a game (with abstract states and moves)

A game implements signature `GAME`. This signature (and its contract) subsumes the contracts for the abstract types `move` and `state`, as well as all the exported functions. The game `state` includes complete information about a game in progress, even "invisible" state like whose turn it is. Each operation's contract is expressed in terms of how it affects states.

The `state` abstraction is immutable—if a mutable representation is chosen, it must be impossible for a client to tell if a mutation has taken place.

```
signature GAME = sig
  structure Move : MOVE
  type state

  (* CREATORS *)

  val initial : Player.player -> state
    (* Initial state for a game in which
       the given player moves first. *)

  (* PRODUCERS *)

  val makemove: state -> Move.move -> state
    (* Returns the state obtained by making the
       given move in the given state.  Raises Move.Move
```

```
      if the state is final or if the given move is not
      legal in the given state. *)

  (* OBSERVERS *)

  val visualize : state -> string
    (* A string representing the given state.
       The string must show whose turn it is,
       and it must end in a newline. *)

  val whoseturn  : state -> Player.player
    (* Given a _non-final_ state, returns the player
       whose turn it is to move.    When given a
       final state, behavior is unspecified. *)

  val isOver : state -> bool
    (* Tells if the given state is final. *)

  val outcome : state -> Player.outcome option
    (* When given a final state, returns SOME applied to the outcome.
       When given a non-final state, returns NONE. *)

  val legalmoves : state -> Move.move list
    (* Lists all moves that are valid inputs to `makemove` in the
       given state.  The list is empty if and only if the given
       state is final. *)

end
```

This interface is not minimal. For example, there are three different ways to tell if a game is over. **They must all agree!**

Third abstraction: a game solver

The `GAME` interface supports a computer opponent based on exhaustive search: an abstract game solver (AGS). An AGS searches possible moves until it finds a move that leads to a win. If it can't find such a move, it tries to force a draw.

The AGS knows nothing about the rules of any game, or even whose turn it is—it knows only which moves can be made in which states, which states are final, and what outcomes are good. It gets all that information from the `GAME` interface. Provided there aren't too many states, the AGS makes a worthy opponent.

The `AGS` interface exports just one function: all a client can do is present a state and ask the AGS what the current player (whose turn it is) should do in that state. If there are any legal moves, the AGS recommends one. And it always says what final outcome is expected, assuming that its recommendation (if any) is followed and that no player ever makes a suboptimal move.

In addition to the `advice` function, the `AGS` contains a complete copy of the game itself! In effect, the `AGS` extends the `Game` with new functionality. In ML, this idiom is common.

```
signature AGS = sig
  structure Game : GAME
  type advice = { recommendation : Game.Move.move option
                , expectedOutcome : Player.outcome
                }
  val advice : Game.state -> advice
    (* Given a non-final state, returns a recommended move,
       plus the expected outcome if the recommendation is followed.
       Given a final state, returns no recommendation,
       plus the outcome that has already been achieved. *)
end
```

The cost model of an AGS is that it tries all possible combinations of moves. AGS functions can be slow. Wait patiently.

Two species of `advice`?!

The `AGS` interface defines a *type* `advice` and also a *value* `advice`. This apparent duplication is possible because Standard ML uses one environment (Γ) for values and a different environment (Δ) for type abbreviations and type constructors. So `advice` stands for *both* a type (in Δ) and a value (the function, in Γ). This idiom is common: the type is what's really important, and the function is named for the type, which is what the function returns.

Exercise G: A game (30%)

Implementing a game will solidify your understanding of the `GAME` interface, which in turn will help you build a good solver. You will choose from these three games:

- In *pick up the last stick*, there are sticks on a table, and players alternate taking 1, 2, or 3 sticks. The player who picks up the last stick wins.
- In *take the last coin*, there is a mix of coins on the table, and players alternate taking coins, but on any turn, all the coins taken have to be of the same denomination. The player who takes the last coin wins.
- In *tic-tac-toe*, players alternate marking squares on a three-by-three grid. The first player to mark three squares in a row wins. (If all squares are marked but neither player has three in a row, the game ends in a tie.)[4]

Each game is more interesting to play (and more time-consuming to implement) than the game before it.

G. Implement a two-player game. Choose a game and implement one of the following:

- Functor `SticksFun` in file `sticks.sml`
- Structure `Coins` in file `coins.sml`
- Structure `TTT` in file `ttt.sml`

Detailed instructions, rules for play, and hints for implementation can be found on the games page. The instructions include the strings that must be recognized by `Move.parse`. **You need these instructions!**

[4] If you want to tackle Tic-Tac-Toe, a good representation of the grid can be created using Standard ML's `Vector` module.

All three games can be played on the departmental servers: run `coins`, `sticks`, or `ttt`.[5] In the first two games, if you play perfectly, you can beat my AGS. In the third, the best you can do is tie.

What to watch out for. We'll test your code to see if your observers provide a consistent picture of a game's state. For example, if your implementation says that player X won, it had better also say that the game is over.

How big are they? Not counting embedded unit tests, my implementations look like this:

- My *pick up the last stick* is 51 lines of Standard ML, but 13 of those lines are blank.
- My *take the last coin* is 107 lines of code and comments, but 23 of those lines are blank. Most of the length is in `visualize` (11 lines) and in `makemove` (12 lines). Depending on what representations you choose, your hot spots will be different.
- My *tic-tac-toe* is 89 lines of Standard ML, of which 5 are blank. Even though it's shorter than my coins game, it was more tiresome to write. (A less clever implementation written by a student at CMU is 144 lines.)

Related reading: The contract of the `GAME` signature, the instructions for the game of your choice on the games page, and the section on ML modules in the ML learning guide.

Test your game with my AGS

Once you're satisfied with your game, you can test it with my AGS. You'll load a functor I've written that instantiates my AGS with your game, then runs the "play manager" with that AGS. Using Moscow ML interactively, follow these steps:

1. Start `mosml` with the options `-I /comp/105/lib -P full`.
2. Load `.uo` files with the `load` command.
3. Use functor application to create a player.
4. Play interactively.

The steps are illustrated by this the annotated transcript below, which uses *take the last coin* as an example:[6]

```
nr@homedog /tmp> mosml -I /comp/105/lib -P full
Moscow ML version 2.10-4 (Tufts University, April 2017)
Enter `quit();' to quit.
- load "playvsnr";   <---- play manager with my AGS
> val it = () : unit
- load "coins";      <---- your game
> val it = () : unit
- structure P = PersonVsAgs(structure Game = Coins);
> structure P : {val go : unit -> outcome}
- P.go ();
Player X sees 4 quarters, 10 dimes, and 7 nickels.
What coins does player X take?
```

If you want to watch the computer play both sides, try this:

[5] All binaries can be found in `/comp/105/bin`.

[6] If you choose the sticks game, you'll have to instantiate your functor `SticksFun` with `val N = 14`.

```
nr@homedog /tmp> mosml -I /comp/105/lib -P full
Moscow ML version 2.10-4 (Tufts University, April 2017)
Enter `quit();' to quit.
- load "playvsnr";   <---- play manager with my AGS
> val it = () : unit
- load "coins";      <---- your game
> val it = () : unit
- structure C = AgsVsAgs(structure Game = Coins);
> structure C : {val go : unit -> outcome}
- C.go ();
Player X sees 4 quarters, 10 dimes, and 7 nickels.
Player X takes [redacted]
```

With this experience in hand, you're ready for the final module of the assignment: the AGS itself.

Exercise A: Abstract Game Solver (40%)

Given a state, an AGS should advise us on the best move *for the player whose turn it is*.

- If the AGS finds a move that enables the current player to force a win, it's done: it recommends that move. It doesn't consider other moves.
- If AGS can't find a winning move, the next best move is one that forces a tie. (In the sticks and coins games, ties are impossible, but ties are a common feature of tic-tac-toe and other games.)
- If the AGS can't win or tie, then all moves lead to losses, and to the AGS, they are all equally bad. It recommends an arbitrary move.

The AGS considers legal moves one by one—but to compute the *consequences* of a move, the AGS calls itself recursively. So the search can be exponential in the length of the game. If you've ever studied AI or search algorithms, you may be aware that there are lots of fancy tricks you can use to cut down the size of a search like this. **Ignore all of them**. Implement the `advice` function in the simplest way possible.

Because the `state` type is abstract, the AGS can't break down the state by forms of data. What the AGS *can* and *should* do is break down *observations*. I recommend breaking down these observations:

- Because the contract of `advice` is itself broken down by cases (one for a final state and one for a non-final state), I recommend that function `advice` begin by calling an observer that tells it whether the given state is final.
- For the case of a non-final state, I recommend breaking the legal moves down by cases. Because the legal moves are a *nonempty* list, you will have two main cases: a singleton list, or a value `move :: moves`, where `moves` is also a nonempty list.[7]
  - If there is just one legal move, the AGS must recommend that move (and whatever outcome the move leads to).
  - If there is more than one legal move, but the first move leads to a perfect outcome (win for the player whose turn it is), the AGS can and should recommend that move, without ever considering the remaining moves.

[7] To make the pattern matching exhaustive, the compiler also requires you to handle the empty list. Since an empty list is impermissible according to the contract for `legalmoves`, I recommend that you handle this case with an expression like "`let exception ContractViolation in raise ContractViolation end`."

- Otherwise the AGS must choose the best move from among those available.

This is *almost* a job for `argMax`; see the third section on implementation tactics.

Tactics for the implementation of the AGS

I've implemented the AGS in three different ways:

1. Continuation-passing style
2. Direct recursion
3. Recursion using something like `argMax`

Continuation-passing style

This is a search problem, so I was tempted to use continuation-passing style. But there's no backtracking, and I needed at least three continuations (one for each possible outcome). I got no benefit from CPS, and the code's a mess. **Not recommended.**

Direct recursion

The key part of `advice` is the search through a *nonempty* list of legal moves. **The design process applies beautifully.** Imagine an internal function `bestAdvice` that takes a nonempty list of moves and returns the advice that is best for the current player. That function might have something like these "mock algebraic laws":

- `bestAdvice [move]` ≡ recommend `move`
- `bestAdvice (move :: moves)` ≡ recommend `move`, when `move` leads to a win for the current player
- `bestAdvice (move :: moves)` ≡ recommend the better of "advise `move`" and `bestAdvice moves`, when `move` does *not* lead to a win for the current player

I found it useful to implement helper functions with these types:

`advice -> outcome`	Extract the outcome from an advice record
`move -> advice`	Advice after making the given move
`outcome -> bool`	Tells if the current player wins
`outcome * outcome -> bool`	Tells if the first outcome is better for the current player
`advice * advice -> advice`	Returns whichever advice is better for the current player

The comparison of the outcomes is annoying, but the rest of the code is nice. **Recommended**.

Recursion using something like `argMax`

The key part of `advice` is the search through a *nonempty* list of legal moves. Each move leads to a configuration, and each configuration can be asked recursively for advice. The AGS should choose the move that maximizes the value of the advice.

This is almost a job for `argMax`. But there's an issue: `argMax` applies the value function to *every* possible move. But when the value function sees a winning move, we shouldn't look any further, because a winning move has the highest possible value.

I defined a version of `argMax` that stops when it hits a ceiling. Because it has to be efficient, it has a type and a contract from hell:

```
val argAtCeilingOtherwiseMax :
  ('a -> 'b) -> ('b -> int) -> int -> 'a list -> 'a * 'b

  (* Calling `argAtCeilingOtherwiseMax f ceiling xs`
     finds an `x` in `xs` such that `f x` is at least
     as big as `ceiling`, or if no such `x` exists,
     an `x` that makes `f x` as big as possible.
     The call returns the pair `(x, f x)`.

     Function `argAtCeilingOtherwiseMax` guarantees
     to call `f` at most once on every element of `xs`.

     The list `xs` must be nonempty.
   *)
```

This function is not pleasant to write or to talk about. But with `argAtCeilingOtherwiseMax` in place, my implementation of `advice` needs only these additional helper functions:

`advice -> outcome`	Extract the outcome from an advice record
`move -> outcome`	Outcome after making the given move (defined by composition)
`outcome -> int`	Value (to the current player) of the given outcome

The resulting `advice` function is super simple, super satisfying. **Recommended with reservations.**[8]

Implement an Abstract Game Solver. Choose one of the implementation tactics and write an AGS in file `ags.sml`. Use the following template:

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

  ... helper functions, if any ...

  fun advice state = ...
end
```

The `where type` declarations are annoying: they look tautological, but they're not. Complain to Dave MacQueen and Bob Harper.

Hints:

- *ML syntax:* The introduction form for the advice record is

[8] Function `argAtCeilingOtherwiseMax` is gnarly, and I can't see a way around it.

`{ recommendation =` e_1`, expectedOutcome =` e_2 `}`

for any expressions e_1 and e_2.

The elimination form is

`let val { recommendation =` x_1`, expectedOutcome =` x_2 `} = …`

for any names x_1 and x_2.[9]

- Do **not** assume that players take turns, that the last player to move always wins, that there are no ties, or any other property of the game you have implemented. Use `whoseturn` and `outcome` instead. We will test your AGS on games that are quite different, including Connect 3 and others. Even a solitaire!
- Unit tests are useful, but they are hard to write *inside* an AGS. If you want unit tests, write them for a particular game. Start with a known configuration and check the two fields returned by `advice`. For example, if a computer player sees a table with only one denomination of coin, its best move is to take all the coins and win. Unit tests like these are game-specific and will have to go into another module. Put them in file `ags-tests.sml`.

To test your AGS, **restart Moscow ML** and `load "ags";`. As long as there is an `ags.uo` in the current working directory, Moscow ML will prefer it to the one we provide in `/comp/105/lib`. You'll be able to run your unit tests. You can also run game-playing integration tests designed for your AGS. You can do this with your own game or one of ours.

How big is it? My CPS version takes about 65 lines of Standard ML. My direct recursion takes under 40 lines. My version using `argAtCeilingOtherwiseMax` takes only 25 lines, but a module exporting `atLeastNOrBestOffer` takes another 20 lines.

Related reading:

- The section on ML modules in the ML learning guide.
- The model solutions for nonempty lists on the `scheme` homework, including the model solution for `arg-max` in µScheme

A common mistake to avoid when debugging your AGS

If you build a simple AGS that fits in 40 lines of code, it is not going to try to fool you: if the AGS cannot force a win, it 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 Tic-Tac-Toe, with you as X, the AGS as O, and play starting in this position:

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

[9] Please choose good names!

You place your X 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, for extra credit you can 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.

How big is it? My aggressive AGS is under 60 lines of Standard ML code.

Test your AGS with my game

Testing your AGS with one of my games requires a slightly different play manager. As an example, I test an AGS with the binary implementation of Tic-Tac-Toe from file `/comp/105/lib/ttt.uo`:

```
nr@homedog /tmp> mosml -I /comp/105/lib -P full
Moscow ML version 2.10-4 (Tufts University, April 2017)
Enter `quit();' to quit.
- load "ags";     <--- your AGS
> val it = () : unit
- load "ttt";     <--- my game
> val it = () : unit
- load "playvsags";  <--- play manager with no AGS
> val it = () : unit
- structure TTTAgs = AgsFun(structure Game = TTT);
> structure TTTAgs : ...
- structure P = PersonVsMyAgs(structure Ags = TTTAgs);
> structure P : {val go : unit -> outcome}
- P.go();
-------------
|   |   |   |
-------------
|   |   |   |
-------------
|   |   |   |
-------------
Player X is to move

Square for player X?
```

If you want to test your own AGS with our Tic-Tac-Toe, our parser recognizes squares `upper left`, `upper middle`, `upper right`, `middle left`, `middle`, `middle right`, `lower left`, `lower middle`, and `lower right`, as well as abbreviations `ul`, `um`, `ur`, `ml`, `m`, `mr`, `ll`, `lm`, and `lr`.

Extra Credit

75c option. Implement a game that is like *take the last coin*, but that has an additional rule: if you take exactly 75 cents' worth of coins, you have the option to move again immediately, before your opponent takes a turn. (As implied by the word "option," the extra move is a choice—a player is never required to exercise the option.) If you choose this extra credit, document your code with an explanation of what you chose to change and why.

Game theory. Professor Ramsey challenges you to a friendly game of "pick up the last stick," with one thousand sticks. The stakes are a drink at the Tower Cafe. As the person challenged, you get to go first. Should you accept the challenge, or should you insist, out of deference to the professor's age and erudition, that the professor go first? Justify your answer.

More. Implement a second game from the approved list.

Even more. Implement all three games on the approved list.

Four. Implement Connect 4.

Aggression. If it can't win, a standard AGS will "give up"—if every move leads to a loss, all moves are equally bad, and it apparently moves at random. That's because a standard AGS assumes that its opponent is perfect. In the dual situation, when the standard AGS knows it can win no matter what, it picks 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 AGS so that it delays losing as long as possible, and when it can win, it wins as quickly as possible. (For example, it should prune the search only if it finds a win on the next move.) One technique is to assign each move a floating-point "benefit" (of type `real`) and to choose the move with the highest benefit.

What and how to submit

Do not copy or submit the signatures provided with the assignment. Submit just the following files:

- A `README` file containing
  - The names of the people with whom you collaborated
  - A list of the exercises that you completed (including any extra credit)
  - The name of the game you chose to implement for credit
  - The names of additional games you might have implemented for extra credit
  - Your answers to the Game Theory extra credit, if you choose to do it
- File `bignum.sml`, implementing the functor which builds signed integers on top of natural numbers (your solution to exercise I)
- A file implementing your solution to exercise G, which must be one of the following:
  - `sticks.sml`
  - `coins.sml`
  - `ttt.sml`
- File `ags.sml`, implementing your solution to exercise A
- File `ags-tests.sml`, containing any unit tests you may have written for your AGS (this file may be empty)
- For exercises G and A, your `coins.mlb`, `sticks.mlb`, or `ttt.mlb` file. (You download this file; for more information see Appendix I.)
- For exercises G and A, any other files you need in order to compile your game and your AGS

The ML files that you submit 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. Don't include copies of files we provide; we already have them, and duplicates can confuse the compiler.

The files you submit must compile with Moscow ML, using the `compile105-sml` script we give you. We will reject files with syntax errors or type errors. Your files must compile *without warning messages*.

As soon as you have the files listed above, run `submit105-sml-pair` to submit a preliminary version of your work. Keep submitting until your work is complete; we grade only the last submission.

Appendices

Appendix I: Two ways to compile Standard ML modules

The *Definition of Standard ML* does not specify where or how a compiler should look for modules in a filesystem. And each compiler looks in its own idiosyncratic way. This issue should be handled by `compile105-sml`, but if something goes wrong, this appendix explains not only what is going on but also how to compile with MLton. (When the time comes to test your game, MLton is going to be important.)

Compiling Standard ML modules using Moscow ML

To compile an individual module using Moscow ML, you type

```
mosmlc -I /comp/105/lib -c -toplevel filename.sml
```

This command 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; don't let it alarm you.

If your module depends on another module, you will have to mention the `.ui` file on the command line as you compile. For example, a `BignumFn` functor depends on both `NATURAL` and `BIGNUM` signatures. If `BignumFn` is defined in `bignum.sml`, `NATURAL` is defined in `natural-sig.sml`, and `BIGNUM` is defined in `bignum-sig.sml`, then to compile `BignumFn` you run

```
mosmlc -I /comp/105/lib -toplevel -c natural-sig.ui bignum-sig.ui bignum.sml
```

The script `compile105-sml` knows about the files that are assigned for the homework, and in most situations it inserts the `.ui` references for you.

Calling `mosmlc` produces a `bignum.uo` file. The `.uo` files are good for two things:

- When you are debugging, you'll can load compiled modules into the interactive system, using `load`. For example,

  ```
  : homework: mosml -I /comp/105/lib -P full
  Moscow ML version 2.10-3 (Tufts University, April 2012)
  Enter `quit();' to quit.
  - load "bignum";
  > val it = () : unit
  ```

  Once you load a module, you cannot recompile it and reload it later. Loading it again has no effect, even if the code has changed; you have to start Moscow ML over again.

- You can use `mosmlc` to link a bunch of `.uo` files together to form an executable binary. For example, the interactive game player can be built like this:

  ```
  mosmlc -I /comp/105/lib -toplevel -o games \
       player-sig.uo player.uo move-sig.uo \
       game-sig.uo ags-sig.uo play-sig.uo \
       slickttt.uo ags.uo aggress.uo coins.uo \
  ```

```
four.uo peg.uo mrun.uo
```

Order matters; 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`.

Compiling Standard ML to native machine code using MLton

If your games are running too slow, compile them with MLton. MLton is a whole-program compiler that produces optimized native code. Because of its superior error messages, MLton can be worth using on other codes, too.

To use MLton, you list all your modules in an MLB file, and MLton compiles them at one go. If you want to try MLton on your own codes, download file `sml-homework.mlb`, then compile with

```
mlton -verbose 1 sml-homework.mlb
```

You can then run any `Unit` tests with `./sml-homework`.[10]

More information about MLton is available on the man page and at `mlton.org`.

If you want to try this with the coins game, download files `coins.mlb` and `runcoins.sml`, and then compile with

```
mlton -output coins -verbose 1 coins.mlb
```

(You can also download and compile `sticks.mlb`, `runsticks.sml`, `ttt.mlb`, and `runttt.sml`.)

Because MLton requires source code, you will be able to use it only after you will have finished your own AGS.

[10] The `mlton-with-unit` script we've been using won't work here, because it's limited to one source file at a time.

Appendix II: How your work will be evaluated

Program structure

We'll be looking for you to seal all your modules. We'll also be looking for the usual hallmarks of good ML structure.

	Exemplary	Satisfactory	Must Improve
Structure	• All modules are sealed using the opaque sealing operator `:>` • Code uses basis functions effectively, especially higher-order functions on list and vector types. • Code has no redundant case analysis • Code is no larger than is necessary to solve the problem.	• Most modules are sealed using the opaque sealing operator `:>` • Code uses the familiar functions, but misses opportunities to use unfamiliar functions like `Vector.tabulate`. • Code has one redundant case analysis • Code is somewhat larger then necessary to solve the problem.	• Only some or no modules are sealed using the opaque sealing operator `:>` • A module is defined without ascribing any signature to it (unsealed) • Code misses opportunities to use `map`, `fold`, or other familiar HOFs. • Code has more than one redundant case analysis • Code is almost twice as large as necessary to solve the problem.

Correctness and performance

We'll look to be sure your code meets specifications, and that the performance of your AGS is as good as reasonably possible.

Exemplary

Satisfactory

Must improve

Correctness

• Game code fulfills the contracts specified in the GAME and MOVE signatures. In particular, every observer and producer presents a consistent view of whether the game is over.

• AGS code makes no additional assumptions about the implementations of Player, Move, or Game.

• Game code fulfills the contracts specified in the GAME and MOVE signatures, except that it permits illegal moves.

• Game code fulfills the contracts specified in the GAME and MOVE signatures, except it sometimes raises the wrong exception.

• Game code violates one of its contracts.

• AGS code assumes that players take turns.

Performance

• The AGS implements its advice function using a single, pruned search that stops once the best move or outcome is known.

• Or, the AGS implements advice by making just one search through the state space of possible game configurations.

• Function advice may search the state space of possible configurations more than once.

• Function advice may search the state space of possible configurations more than twice.