Modules and Abstract Types

COMP 105 Assignment

Due Thursday, April 20, 2017 at 11:59PM

Contents

Overview

In this assignment, you will

- Learn about abstract data types
- Learn about arbitrary-precision arithmetic, which you will see again
- Use Standard ML modules to write client code for an interface that is not yet implemented
- Use Standard ML modules to put together a nontrivial program
- See how to reuse code that depends not just on other values, but also on other types

You'll accomplish these ends by completing the four parts of the assignment:

- In reading comprehension, you read about new language features (ML modules), new ideas (abstraction function and representation invariant), and new algorithms (arithmetic)
- In the problem Q, you write client code that uses a priority queue—without an implementation. In effect, you implement heapsort despite not having a heap. This part involves a lot of reading and understanding what is being presented and asked. There is just a tiny amount of programming.
- In problem N, you implement arbitrary-precision arithmetic on natural numbers. These so-called "bignums" are an important programming-language abstraction, and they are an ideal problem for exploring the freedom of choice offered to you by abstract types. Substantial thought and substantial programming are required.
- In problem A, you implement an unsophisticated (yet unbeatable) computer opponent that can be used to play many different two-player games. To help you understand and test this very popular problem, you will also implement problem S: an unsophisticated game called "pick up the last stick." The computer opponent requires careful thought but not a lot of code: in essence, it boils down to one carefully crafted recursive function. The simple game requires almost no thought, and not a lot of code—though it will seem like a lot, because it is all boilerplate.

As usual, you do reading-comprehension questions on your own. You may tackle the remaining problems either on your own or with a partner.

Setup

The code in this handout is installed for you in `/comp/105/lib`, where you don't have to look at it. You will compile your own code using a special script, `compile105-sml`, which is available on the servers through the usual command `use comp105`. This script does not produce any executable binary. Instead,

it creates binary modules ("`.uo` files") that you can load into Moscow ML, as in `load "ags";`. You can use it to compile all files or just a single file:

```
compile105-sml
compile105-sml ags.sml
```

To be able to load the binaries that we provide, you must supply an additional argument to `mosmlc` and `mosml`, as in

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

Reading comprehension

These problems will help guide you through the reading. We recommend that you complete them before starting the other problems below. You can download the questions[1].

1. Using one of the sources in the ML learning guide[2], read about structures, signatures, and matching. Then answer questions about the structure and signature below.

   The following structure contains definitions that should be familiar from the ML homework[3] and from code you may have seen in the course interpreters:

   ```
   structure ExposedEnv = struct
     type name   = string
     type 'a env = (name * 'a) list
     exception NotFound of name
     val emptyEnv = []

     fun lookup (name, [])              = raise NotFound name
       | lookup (name, (x, v) :: pairs) =
           if x = name then v else lookup (name, pairs)

     fun bindVar (name, value, env) = (name, value) :: env
   end
   ```

   Here is a signature:

   ```
   signature ENV = sig
     type name = string
     type 'a env
     val emptyEnv : 'a env
     val lookup   : name * 'a env -> 'a
     val bindVar  : name * 'a * 'a env -> 'a env
   end
   ```

   Answer these questions:

   (a) Does the signature match the structure? That is, if we write

[1] ./cqs.sml.txt
[2] ../readings/ml.html
[3] ml.html

```
structure Env :> ENV = ExposedEnv
```

does the resulting code typecheck?

(b) Does the signature expose enough information for us to write the following function? Explain why or why not.

```
fun extendEnv (names, vals, rho) =
  ListPair.foldrEq Env.bindVar rho (names, vals)
```

(c) Does the signature expose enough information for us to write the following function? Explain why or why not.

```
fun isBound (name, rho) = (Env.lookup (name,rho) ; true)
                          handle Env.NotFound _ => false
```

(d) If in part (b) or part (c), it is not possible to write the function given, explain how you would change the code to make it possible.

(e) Suppose you change the `ENV` signature to make the `name` type abstract, so the signature reads

```
signature ENV = sig
  type name
  type 'a env
  ...
end
```

Is more abstraction better here? Or does the change have unfortunate consequences? Justify your answer.

You now have the basic ideas needed to understand what is being asked of you in this assignment, and you know enough to implement most of the "pick up the last stick" game (problem S).

2. An ML *functor* is a function that operates on the module level. Its *formal* parameters, if any, are specified by a *sequence of declarations*, and its *actual* parameters are given by a *sequence of definitions*. Its result is a structure. Read about functors in Harper, as recommended in the ML learning guide.

   Here's a typical application of functors. To keep track of the thousands of tests we run on students' code, I need an efficient "test set" data structure. But not all tests have the same type. To reuse the data structure with tests of different types, I need a functor. Here is what my "test set" functor needs to know about a test:

   - A string identifying the student who wrote it
   - A comparison function that provides a total order on tests
   - A function that converts a test to a string, for printing

   Using this information, answer parts (a) and (b):

   (a) Write down the information needed for a test set in the form of *formal parameters* for the functor `TestSetFun`, keeping in mind that a functor's formal parameters are written as a sequence of declarations:

```
functor TestSetFun(
                  ... fill in formal parameters here ...
                )
```

```
:> TEST_SET = struct ... end  (* ignore this part *)
```

(b) Now focus your attention on one particular test, the `check-type` test:

- The test is identified by a student's user ID and a sequence number 1, 2, or 3.
- The test contains an expression of type `exp` and a type of type `ty`.
- Comparison can be implemented using code like this:

```
case String.compare (uid1, uid2)
  of EQUAL => Int.compare (seqno1, seqno2)
   | diff  => diff
```

- A test can be converted to a string code like this:

```
concat ["(check-type ", expString e, " " , tyString tau, ")"]
```

Show how to create a set of `check-type` tests by filling in the *actual parameters* for the `TestSetFun` functor:

```
structure CheckTypeSet :> (TEST_SET where type test = check_type_test)
  =
TestSetFun(
           ... fill in actual parameters here ...
          )
```

You now understand functors well enough to use them in problems 1 and 2.

3. Read about "signature refinement or specialization" in the ML learning guide[4]. Now,

   (a) Explain what, in part (b) of the previous question, is going on with the `where type` syntax.

   (b) Explain what would go wrong if we wrote this code instead:

   ```
   structure CheckTypeSet :> TEST_SET = TestSetFun(...)
   ```

   You now understand functors well enough to know how to refine the result signature of your Abstract Game Solver in problem A.

4. Many useful abstractions can be implemented using the humble binary tree. Let's consider how we might use a *binary-search tree* to store a large data structure containing all of the software testing results from 105. I want to know for each test, for each student, did the student's submission pass the test.

   Read Ramsey, section 8.1, on abstractions, representations, and invariants. Focus on places where binary trees are mentioned. Also read the first three pages or section 8.8.4 on hash tables, which represent a similar abstraction. (Skip all the code parts.)

   (a) In informal English ("the world of ideas"), what is the abstraction represented by the binary-search tree?

   (b) Suppose a binary-search tree is represented as follows:

   ```
   datatype tree = EMPTY
                 | NODE of tree * (test * student * bool) * tree
   ```

[4] ../reading/ml.pdf

Using some combination of ML code and informal English, write the *abstraction function* which maps this representation onto the abstraction that it stands for:

(c) Using ML code, and assuming whatever functions you need on types `test` and `student`, write the *representation invariant* of the binary-search tree. (The representation invariant of a binary-search tree is the *order* invariant which guarantees the correctness of `insert` and `lookup`.)

You now know enough to design an abstraction function and representation invariant for your chosen representation of natural numbers in problem 2.

5. In "Mastering Multiprecision Arithmetic"[5], read the sections on addition, subtraction, and multiplication.

   - Given that you may be adding two numbers of different lengths (e.g., 1777944 + 903), what sort of case analysis do you expect to need to implement addition?
   - Given that you may be subtracting two numbers of different lengths (e.g., 1777944 - 903), what sort of case analysis do you expect to need to implement subtraction?
   - Given that you may be multiplying two numbers of different lengths (e.g., 1777944 * 903), what sort of case analysis do you expect to need to implement multiplication?

   You are now prepared to tackle the implementation part of problem 2.

Programming problem Q: The module warmup

Q. Abstract data type: priority queues. A *priority queue* is a collection of *elements* with an operation that quickly finds and removes a minimal element. (The priority queue assumes that a total order exists on the elements; a minimal element is an element of the priority queue that is at least as small as any other element. A priority queue may contain more than one minimal element, in which case it is not specified which such element is removed.)

Like most data structures, priority queues come in both immutable and mutable forms. Immutable data structures are easier to test, specify, and share. Mutable data structures usually have lower space costs and sometimes lower time costs. Here is an interface that describes an *immutable* priority queue:

```
signature PQUEUE = sig
  type elem  (* element (a totally ordered type) *)
  val compare_elem : elem * elem -> order (* observer: total order of elems *)

  type pqueue  (* Abstraction: a sorted list of 'elem' *)
  val empty  : pqueue                   (* the empty list *)
  val insert : elem * pqueue -> pqueue  (* producer: insert (x, xs) == sort (x::xs) *)
  val isEmpty : pqueue -> bool          (* observer: isEmpty xs == null xs *)
  exception Empty
  val deletemin : pqueue -> elem * pqueue
    (* observer: deletemin [] = raises the Empty exception
                 deletemin (x::xs) = (x, xs) *)
```

[5] ../readings/arithmetic.pdf

```
  (* cost model: insert and deletemin take at most logarithmic time;
                  isEmpty takes constant time*)
end
```

The type `pqueue` is abstract, so its representation is not specified, but the interface says what abstraction a `pqueue` corresponds: a sorted list of values of type `elem`.[6] Each operation is described by giving the *imagined* effect on the abstraction. When specifying an interface that uses abstract data types, working with an abstraction in this way is a best practice—but not always easy or even possible.

Here is a mutable version of the same abstraction:

```
signature MPQUEUE = sig
  type elem  (* element (a totally ordered type) *)
  val compare_elem : elem * elem -> order (* observer: total order of elems *)

  type pqueue  (* Abstraction: mutable cell containing a sorted list of 'elem' *)
  val new : unit -> pqueue              (* returns `ref []` *)
  val insert : elem * pqueue -> unit    (* q := sort (x :: !q) *)
  val isEmpty : pqueue -> bool          (* observer: isEmpty xs == null xs *)
  exception Empty
  val deletemin : pqueue -> elem        (* remove and return the first element *)
end
```

I can conclude that the abstraction is mutable just by inspecting the types: I see `unit` types, and I observe that `deletemin` doesn't return a `pqueue`.

Programming problem Q. Use a priority-queue abstraction to implement a sort module matching this signature:

```
signature SORT = sig
  type elem
  val compare : elem * elem -> order
  val sort : elem list -> elem list
end
```

You may use either of the abstractions defined above. Your choice of abstraction will determine the details of your solution:

- If you choose the immutable abstraction, write a functor `PQSortFn` that takes a structure matching signature `PQUEUE` and produces a structure matching `SORT`, like this:

  ```
  functor PQSortFn(structure Q : PQUEUE) :> SORT where type elem = Q.elem =
  struct
    ...
  end
  ```

- If you choose the mutable abstraction, write a functor `MPQSortFn` that takes a structure matching signature `MPQUEUE` and produces a structure matching `SORT`, like this:

  ```
  functor MPQSortFn(structure Q : MPQUEUE) :> SORT where type elem = Q.elem =
  struct
  ```

[6] Sometimes a priority queue is said to correspond to a "bag" of elements or even a set, but a sorted list is simpler.

```
  ...
end
```

Either way, place your solution in file `sort.sml`. You can compile it by running

```
compile105-sml sort.sml
```

You need not implement `PQUEUE` or `MPQUEUE`. This is the whole point! (Just keep in mind that without an implementation, you cannot run unit tests.)

Also, you do not need the source code that defines the signatures `PQUEUE` and `MPQUEUE`—the `compile105-sml` script finds those files in `/comp/105/lib`.

How big is it? The body of the functor is under 10 lines of code.

Related reading:

- Read the brief discussion of the priority-queue abstraction on page 538 of Ramsey.
- If you wish, you may model your sort on the heapsort presented in Section 8.8.2, which starts on page 594. Instead of CLU's `for` loop, you can use a higher-order function.
- Read about functors in sources from the ML learning guide.

What's the point? This problem illustrates a key advantage of the ML module system: you can program against an *unimplemented* interface. When you're designing an interface, you often need to know if the interface meets the needs of its clients, and you want to get the design right *before* you spend the budget needed to create an implementation. Several languages provide mechanisms that will do the job (e.g., Java interfaces); in ML, those mechanisms are signatures and functors.

Programming problem N: Arbitrary-precision natural numbers

Standard ML 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 implementation raises `Overflow`. In this problem, you implement natural numbers—or if you prefer, unsigned integers—of arbitrary precision. Arithmetic will *never* overflow; the worst that can happen is to run out of memory.

You will implement this interface:

```
signature NATURAL = sig
   type nat
   exception Negative   (* raised if the result of any operation would be negative *)

   exception BadDivisor

   val of_int : int -> nat          (* could raise Negative *)
   val /+/    : nat * nat -> nat
   val /-/    : nat * nat -> nat    (* could raise Negative *)
   val /*/    : nat * nat -> nat
   val sdiv   : nat * int -> { quotient : nat, remainder : int }
```

```
    (* Contract for "Short division" sdiv:

       Provided 0 < d <= base,
         sdiv (n, d) returns { quotient = q, remainder = r }
       such that
         n == q /*/ of_int d /+/ of_int r
         0 <= r < d

       Given a d out of range, sdiv (n, d) raises BadDivisor

       We know that base >= 10, but otherwise its value is unspecified
    *)

  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 (of_int 123) = [1, 2, 3]
                     decimal 0            = [0]
       It must never return an empty list.
    *)
end
```

N. Abstract data type: natural numbers. You will design and build your implementation in two parts.

(a) **Abstraction function and representation invariant**.

A natural number should be represented by a sequence of digits. But this still leaves you with interesting design choices:

- Do you prefer an array or a list?
- If you prefer an array, do you want a mutable array (type `'a array`) or an immutable array (type `'a vector`)?
- If you prefer a list, do you prefer to store the digits of a number in big-endian order (most significant digit first) or little-endian order (least significant digit first)?

  (If you prefer an array, the only order that makes sense is to store the least significant digit in slot 0, and in general, to store the digit that is multiplied by b^n in slot n.)

Some implementations of arithmetic operations, especially multiplication, can leave leading zeroes in a representation. The number of leading zeroes must be controlled so that it doesn't grow without bound.

- Do you prefer a representation invariant the eliminates leading zeroes?
- If not, how will you track the number of leading zeroes in each representation? Will you store it in the representation itself, or will you compute it dynamically?

The beauty of abstract data types is that *the interface isolates these decisions, so you can easily change them*.

In part (a) of this problem, you pick a representation and an invariant. You document them by writing an abstraction function and representation invariant. Do so by completing this template:

```
functor NaturalRep (structure N : NATURAL) :>
  sig
    datatype digit = D of int
    type rep = ...  (* your choice of representation *)
    val invariant   : rep -> bool
    val abstraction : rep -> N.nat
  end
=
  struct
    datatype digit = D of int
    type rep = ...
    exception LeftAsExercise of string
    val base = raise LeftAsExercise of string
    fun invariant   r = raise LeftAsExercise "invariant"
    fun abstraction r = raise LeftAsExercise "abstraction"
  end
```

Place your solution in file `natural-rep.sml`. Compile by

```
compile105-sml natural-rep.sml
```

Put a note in your README saying which representation you picked and why.

Hints:

- No one representation is equally good for all operations. You can pick one representation and stick with it, or you can convert temporarily as needed to facilitate each operation. You can even define a representation as a set of alternatives, as in

  ```
  datatype nat = ARRAY         of digit array
               | LITTLE_ENDIAN of digit list
               | BIG_ENDIAN    of digit list
  ```

  A little freedom can be dangerous: if you choose a representation like this one, you risk having to handle at least nine cases per binary operator. But the choice is yours.

- As you complete part (b), you may want to revisit your decisions. That's part of the point—representations can change without affecting any external code.

Related reading:

- Read about "Choice of representation" on the first page of the handout "Mastering Multi-precision Arithmetic"[7]
- Read the algebraic formulation of arithmetic in Section 8.9.2, which starts on page 604 of Ramsey, and look at the examples in that section.

[7] ../readings/arithmetic.pdf

- Read about *representation invariants* and *abstraction functions* on page 584 of Ramsey. If you want more depth, the first part of Section 8.7, which starts on page 582, may be useful. Don't bother with sections 8.7.1, 8.7.2, or 8.7.3.
- By this point, I expect you're solid on ML signatures, structures, and functors. If not, there's the ML learning guide[8].

(b) **Implementation of all the operations**.

In file `natural.sml`, write a structure `Natural` that implements signature `NATURAL`. The right way to do it is with code that looks like this:

```
(* inconvenient code *)
structure Natural :> NATURAL = struct
  ...
  type nat = ...  (* abstraction: ... *)
                  (* invariants: ... *)
  ...
end
```

But there's a small problem here: once the module is sealed, you will find it almost impossible to debug. Here's a trick of the ML masters:

```
(* convenient code *)
structure ExposedNatural = struct               (* Look!  No ascription *)
  ...
  type nat = ...  (* abstraction: ... *)
                  (* invariants: ... *)
  ...
end

structure Natural :> NATURAL = ExposedNatural   (* the module is sealed here *)
```

You can debug `ExposedNatural`, while your client code uses the sealed `Natural`.

Whichever way you choose to write it, be sure you seal structure `Natural` with the `:>` operator, so that no client code can see the representation and violate its invariants. Sealing is especially important if you choose a mutable representation of this immutable abstraction.

Be sure also to document your abstraction function and representation invariant here also. A couple of informal sentences will do nicely.

How big is it? I wrote two implementations: one array-based and one list-based. Together they total about 270 lines of code, of which 50 of those lines are blank. The two implementations are roughly the same size.

Related reading:

- The short handout "Mastering Multiprecision Arithmetic"[9] recommends exactly how to implement natural numbers, including what helper functions you will find most useful and what operations to implement first.

[8] ../readings/ml.pdf
[9] ../readings/arithmetic.pdf

- You will find many hints in Exercise 39 on page 636.
- For addition and subtraction, read Ramsey, page 605; and Hanson, pages 305 to 307.
- For multiplication, read Ramsey, page 606; and Hanson, pages 307 and 308.
- For short division, read Ramsey, pages 606 and 607; and Hanson, pages 311 and 312 (the `XP_quotient` function). What Ramsey calls a "remainder" r_i is called `carry` by Hanson.
- By this point, I expect you're solid on ML signatures, structures, and functors. If not, there's the ML learning guide[10].

Mistake to avoid: Don't try to multiply by repeated addition—multiplication of large numbers won't terminate in your lifetime. You must multiply each pair of digits (one each from the multiplicand and the multiplier) and add up the partial products, appropriately shifted.

What's the point? The key point here is that abstract data types put a firewall between an interface and its implementation, so that you can easily change the base of natural numbers, or even the representation, and no program can tell the difference. In Standard ML, this degree of abstraction is achieved through opaque signature ascription, known to a select group of 105 students as "the beak."

Testing arbitrary-precision arithmetic

To be able to test anything at all, you need `of_int` and `decimal`, which probably also means `sdiv`. If you can't get `sdiv` working and you're desperate to write tests, try using base 10 and writing a special-purpose implementation of `decimal`.

With these in place, compute some 20-digit numbers by taking a list of digits and folding with multiply and add. Since you only have to multiply by 10, you can test addition without multiplication. Here's a function that multiplies by 10:

```
fun tenTimes n =
  let infix 6 /+/
      val p = n
      val p = n /+/ n   (* p ==  2n *)
      val p = p /+/ p   (* p ==  4n *)
      val p = p /+/ n   (* p ==  5n *)
      val p = p /+/ p   (* p == 10n *)
  in  p
  end
```

Once you are confident that addition works, you can test subtraction of natural numbers by generating a long random sequence, then subtracting the same sequence in which all digits except the most significant are replaced by zero.

You can create more ambitious tests of subtraction by generating random natural numbers and using the algebraic law $(m + n) - m = n$. You can also check to see that unless n is zero, $m - (m + n)$ raises the `Negative` exception.

It is harder to test multiplication, but you can at least use repeated addition to test multiplication by *small* values.

[10] ../readings/ml.pdf

You can also easily test multiplication by large powers of 10.

Beyond these simple kinds of tests, civilized people don't write unit tests by hand—we write computer programs that generate unit tests. At any point, you can just generate random expressions that compute with large numbers, then compare your results with a working implementation of bignums. Both MLton and Standard ML of New Jersey (command `sml`) provide a structure `IntInf` that implements arbitrary-precision *signed* integers.

I have provided some computer-generated tests[11].

Understanding and representing adversary games

In problems S and A 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.

Separating 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[12] and modified by Norman Ramsey.)

The `GAME` signature declares all the types and functions that an Abstract Game Solver 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. (I have even implemented 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. Think of a configuration as a marked-up tic-tac-toe board, or even just a number of sticks sitting on a table. 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;

[11] ./natural-tests.sml

[12] http://www.cs.berkeley.edu/~necula/

the idea is to stop or "prune" the search before it goes too far. Many advanced pruning techniques have been developed for solving games, and if you wish, you are welcome to try them, but for this assignment, you don't have to—exhaustive search works really well.

Basic data in the problem: Players and outcomes

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

- There are two *players*.
- 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`:

```
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
```

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`.

```
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 = "a tie"
    | outcomeToString (WINS p) = toString p ^ " wins"
end
```

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 an abstract game: the `GAME` signature

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.

```
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.
                     Configurations must appear immutable.
                     If a mutable representation is used, 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 one player has won,
                   or it might be that nobody can move
                   (which would be considered a tie). *)

  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
```

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

Specification of a game solver: the `AGS` signature

A solver for a `GAME` exports only two new functions:

- Function `bestmove` returns the best available move in any configuration. "Best" is always from the point of view of the player whose turn it is. If no moves are available—that is, if the configuration is final—`bestmove` returns `NONE`. The computer player uses the result from `bestmove`.
- Function `forecast` looks at a configuration and predicts what the outcome will be if both players make perfect moves. It is useful for testing.

In addition to these functions, 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
  val bestmove : Game.config -> Game.Move.move option
                          (* Given a configuration, returns the
                           * most beneficial move for the player
                           * to move *)
```

```
  val forecast : Game.config -> Player.outcome
                          (* Given a configuration, returns the
                           * best possible outcome for the player
                           * whose turn it is, assuming opponent
                           * plays optimally *)
end
```

The cost model of the AGS is that it tries all possible combinations of moves. For some games, the AGS functions are slow. Be patient.

Programming problem S: Implement "take the last stick"

The main foci of this assignment are the natural numbers and the Abstract Game Solver. But to understand how the solver works, it helps you to implement a simple two-player game. You'll implement a very ~~boring~~ *easy* game called "pick up the last stick." Here are the rules:

- The game starts with `N` sticks on a table. We require $N = 14$.
- Players take turns.
- When it's your turn, you must pick up 1, 2, or 3 sticks.
- The player who picks up the last stick wins.

Put the code for your game into a file called `sticks.sml`, which be organized according to the following template:

```
functor SticksFun(val N : int) :> GAME = struct
  ...
  type config = ...   (* or possibly    datatype config = ... *)
  ...
  structure Move = struct
     datatype move ...    (* pick one of these two ways to define type 'move' *)
     type move = ...
     ...
  end
  ...
end
```

Complete the implementation by following these step-by-step instructions for implementing two-player games:

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. You want a representation that will make it easy to implement `makemove`, `possmoves`, and `outcome`.

   In "pick up the last stick," there's an obvious choice: a pair containing the player who's turn it is and the number of sticks on the table. For a more ambitious game, like Hexapawn[13] or tic-tac-toe,

[13] https://en.wikipedia.org/wiki/Hexapawn

the representation is less obvious. I myself have implemented tic-tac-toe in two different ways, one of which outperforms the other by a factor of four.

You might be tempted to use mutable data to represent a 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.

b. Choose a representation for moves. That is, define type `move`. Everything said for configurations applies here also, but this choice seems less critical.

   For "pick up the last stick", there are two good choices: you could pick type `int`, in which case you'd have to enforce a representation invariant, or you could define an enumeration type such as

   ```
   datatype move = ONE | TWO | THREE
   ```

   The choice is yours.

c. Define the exception `Move`.

d. Write function `initial`.

e. Write 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 "pick up the last stick":

   - There are no ties.
   - The game is over only if there are no sticks left on the table.
   - If there are no sticks left, the player whose turn it is *loses*.

h. Write `finished`. This function should return true if somebody has won or if no move is possible (everybody is stuck).

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`.)

k. Write `Move.toString`. This function must return a string like "Player X picks up 1 stick" or "Player O moves to the middle square." The string must *not* contain 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`.

l. 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.

   `Move.toString` and `toString` don't affect the AGS; they are used by the interactive player to show you what's happening.

m. 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.

n. 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.

   So that we can test your code, please ensure that `Move.fromString` accepts either the strings `"1"`, `"2"`, and `"3"` or the strings `"one"`, `"two"`, and `"three"`.

The "take the last stick" game is so simple that it doesn't need a lot of unit testing, but it is probably worth writing a couple of tests. When people implement two-player games, their most common mistake to permit players to continue to move even when the game is over—if I had any concerns about correctness, I would focus on tests to ensure that `possmoves`, `finished`, and `outcome` are all consistent. Use the Unit[14] signature from `/comp/105/lib`; the `compile` script should import it for you.

How big is it? Not counting embedded unit tests, my implementation is 43 lines of Standard ML. But ten of those lines are blank. The median function is 2 lines long, and no single function takes more than 6 lines of code.

Related reading: The lengthy description of the `GAME` signature, and the section on ML modules in the ML learning guide[15].

What's the point? Parametric polymorphism on a large scale requires something like modules. The main point of implementing the stick game is to enable you to understand games well enough to write an AGS.

Integration testing, part I: your game with my AGS

Once you're satisfied with your game, you can test to see how it works when combined with my AGS as a computer player. You will create an instance of your game with $N = 14$, use the instances to create a game-specific AGS, then use that instance with the computer player. All this will be done interactively using Moscow ML. The key steps are as follows:

- Start `mosml` with the options `-I /comp/105/lib -P full`.
- Load `.uo` files with the `load` command.
- Use functor applications to create the components you need.
- Play interactively.

Here is an annotated transcript:

```
: homework> mosml -I /comp/105/lib -P full
Moscow ML version 2.10-3 (Tufts University, April 2012)
Enter 'quit();' to quit.
- load "sticks";         <---- your game
> val it = () : unit
- load "ags";            <---- my AGS
> val it = () : unit
- structure Sticks14 = SticksFun(struct val N = 14 end);   <--- create Game structure
> structure Sticks14 : ...
- structure S14Ags = AgsFun(structure Game = Sticks14);    <--- create Ags structure
```

[14] Unit.sig.txt
[15] ../readings/ml.pdf

```
> structure S14Ags : ...
```

Once you have your game-specific AGS, you create an interactive player by applying functor `PlayFun` to your AGS. To get `PlayFun`, load file `play.uo`:

```
- load "play";
> val it = () : unit
- structure P = PlayFun(structure Ags = S14Ags);   <--- create the player
> structure P : ...
```

Functor `PlayFun` returns a structure that implements the following signature:

```
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
```

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). Here are some suitable definitions.

```
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 = Sticks14.initial Player.X
      (* Empty configuration with X to start *)

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

With these definitions in place, you can start a game:

```
- P.play computero cnfi;
Player X sees | | | | | | | | | | | | | |
How many sticks does player X pick up?
```

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

```
- P.play computerxo cnfi;
Player X sees | | | | | | | | | | | | | |
```

```
Player X takes [redacted] sticks
...
```

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

Programming problem A: Build an Abstract Game Solver

A. Implement an Abstract Game Solver. Given a configuration, an AGS should pick the best move:

- If the AGS finds a move that enables the current player to force a win, it's done: it picks that move. It doesn't even have to consider other moves.
- If AGS can't find a winning move, the next best move is one that forces a tie. (In the stick game, ties are impossible, but they 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.

The AGS looks at moves by simple linear search—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**. Instead, build your AGS around this helper function:

```
val bestresult : Game.config -> Game.move option * result
```

where `result` is a representation you choose. The idea is that `bestresult conf = (bestmove, whathappens)` where

- If the player can't move, `bestmove` is `NONE`.
- If the player can move, `bestmove` is `SOME m`, where `m` is the best possible `Game.move` for the player in this configuration.
- Value `whathappens` explains what the AGS predicts is the outcome of the game if both players play perfectly. It suffices to use a result 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.

  The `whathappens` value is computed inductively. In the base case, `conf` is a finished configuration, and `whathappens` is determined by the return value from `Game.outcome`. In the inductive step, the AGS *chooses* `whathappens` by recursively evaluating the best possible result from each move. It then picks the result from the move that is best for the current player.

To compare outcomes, I recommend creating a helper function that calculates, for any given `result`, the *benefit* that the result provides to the current player. A win confers maximum benefit; a tie confers medium benefit; and a loss confers minimum benefit. There are more sophisticated ways to view benefits; for example, we could assign larger benefits to winning quickly, and so on. But distinguishing wins, losses, and ties is good enough.

Write an AGS using the following template:

```
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
```

Note how annoying the `where type` declarations are: they look tautological, but they're not. Complain to Dave MacQueen[16] and Bob Harper[17].

Hints:

- You can implement the inductive step by using `map` with the result of `Game.possmoves`. But if you use this code, your AGS will always search *every* possible move, even if it finds a winning strategy on the very first move. This code will make your AGS slow and no fun to play. Instead, write a simple recursive function so that if the AGS finds a move that confers maximum benefit, it can halt the search right away and just return the good move. It will take just a few lines of code, and you will have a lot more fun.
- 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 "pick up the last stick." Use `whoseturn` and `outcome` instead. We will test your AGS on games that are quite different from "pick up the last stick", including Tic-Tac-Toe, Connect 3, and others. Probably even a solitaire!
- It is hard to write unit tests *inside* an AGS. If you want unit testing, write unit tests for a particular game. Start with a known configuration and check `forecast` and `bestmove`. For example, if a computer player sees a table with two sticks, its best move is to pick up both sticks 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, all you need to do is restart Moscow ML and once again `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, as well as the same kind of game-playing integration tests you used wih your stick game. If you want to test your AGS with a more sophisticated game, try our tic-tac-toe game, as described below.

How big is it? My AGS takes about 40 lines of Standard ML.

Related reading: the section on ML modules in the ML learning guide[18].

What's the point? The AGS requires one short but subtle recursive function, and it presents a simple, narrow interface. But look at the parameters! To try to implement an AGS using parametric polymorphism, you would need at least two type parameters (configuration and move), and you would need at least four function parameters (`whoseturn`, `makemove`, `outcome`, and `possmoves`). Writing the code would become very challenging—polymorphism and higher-order programming at the value level is the

[16] http://people.cs.uchicago.edu/~dbm/
[17] http://www.cs.cmu.edu/~rwh/
[18] ../readings/ml.pdf

wrong tool for the job. The point of this exercise is to use a functor to take an *entire* `Game` structure at one go: both types and values. Moreover, the *same* `Game` structure also drives the computer player and the interactive player—nothing in the `Game` module has to change. Programming at scale requires some sort of tool that bundles types and code into one unit.

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 |   |
-------------
|   |   |   |
-------------
```

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, 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.

Integration testing, part II: your AGS with my game

We supply a binary implementation of Tic-Tac-Toe in file `/comp/105/lib/ttt.uo`. You can use it as follows:

```
homework> mosml -I /comp/105/lib -P full
Moscow ML version 2.10-3 (Tufts University, April 2012)
Enter 'quit();' to quit.
- load "ags";
> val it = () : unit
- load "ttt";
> val it = () : unit
- structure TTTAgs = AgsFun(structure Game = TTT);
> structure TTTAgs : ...
- structure PT = PlayFun(structure Ags = TTTAgs);
> structure PT : ...
```

```
- PT.play (PT.getamove [Player.O]) (SlickTTT.initial Player.X);
-------------
|   |   |   |
-------------
|   |   |   |
-------------
|   |   |   |
-------------
Player X is to move

Square for player X?
```

Descriptions of two-player games

Here are descriptions of 4 games: "Pick up the last stick," "Tic-Tac-Toe," "Nim," and "Connect 4". Do not worry if you haven't seen these games before—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.

Pick up the last stick

The game starts with `N` sticks on a table. Players take turns. When it's your turn, you must pick up 1, 2, or 3 sticks. The player who picks up the last stick wins.

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). In the initial configuration, the board is empty.

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 Britain, this game is called "noughts and crosses." No matter what you call it, a player who plays perfectly cannot lose. All your base are belong to the AGS. You can play `/comp/105/bin/ttt`.

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 are 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. You can play `/comp/105/bin/nim`.

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. You can play `/comp/105/bin/four`.

Extra Credit

Proof. Prove that any of these simple games is always in one of these three states:

- The player whose turn it is can force a win.
- Either player can force a tie.
- The player whose turn it is can be forced to lose.

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.

Tic-tac-toe. Implement tic-tac-toe.

Four. Implement Connect 4.

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`.

What and how to submit

As soon as you have tackled the reading comprehension, run `submit105-sml-solo` to **submit your answers to the CQ's**.

For the programming problems, submit the following files:

- A `README` file containing
  - The names of the people with whom you collaborated
  - A list of the problems that you solved (including any extra credit)
  - A short statement explaining which representation you picked for natural numbers, and why
  - Answers to Proof and Game Theory extra credits.
- File `sort.sml`, implementing your solution to Problem Q
- Files `natural-rep.sml` and `natural.sml`, implementing your solutions to problem N
- File `sticks.sml`, implementing your solution to Problem S
- File `ags.sml`, implementing your solution to Problem A
- File `ags-tests.sml`, containing any unit tests you may have written for your AGS
- For problems S and A, your `compile` and `sticks.mlb` files. (See Appendix 1 for information on the these files.)
- For problems S and A, any other files you need in order to compile `sticks.sml` and `ags.sml`.

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. 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 must 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.

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.

Acknowledgments

The AGS assignment is derived from one graciously provided by Bob Harper[19]. George Necula[20], 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.

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. You should be able to get away with using `compile105-sml`, but if something goes wrong, this appendix explains not only what is going on but also how to compile with MLton.

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 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 `DictFn` functor depends on both `DICT` and `KEY` signatures. If `DictFn` is defined in `dict.sml`, `KEY` is defined in `key-sig.sml`, and `DICT` is defined in `dict-sig.sml`, then to compile `DictFn` you run

```
mosmlc -I /comp/105/lib -toplevel -c dict-sig.ui key-sig.ui dict.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.

[19] http://www.cs.cmu.edu/~rwh/

[20] http://www.cs.berkeley.edu/~necula/

To talk about what happens after you compile, I'll use another example:

```
mosmlc -I /comp/105/lib -c -toplevel /comp/105/lib/game-sig.ui /comp/105/lib/player.ui sticks.sml
```

This compilation produces two files:

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

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

- When you are debugging, it is tremendously useful to get compiled modules into the interactive system. Load them directly using `load`, e.g.,

  ```
  : homework: mosml -I /comp/105/lib -P full
  Moscow ML version 2.10-3 (Tufts University, April 2012)
  Enter 'quit();' to quit.
  - load "sticks";
  > val it = () : unit
  - structure Sticks12 = SticksFun(struct val N = 12 end);
  > ...
  ```

  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. To do anything interesting, one of the source files should have a top-level call to `play`, `forecast`, or some other interesting function.

  Here is an example of a command line I use on my system to build an interactive game player:

  ```
  mosmlc -I /comp/105/lib -toplevel -o games \
                            player-sig.uo player.uo game-sig.uo \
                            ags-sig.uo play-sig.uo slickttt.uo \
                            ags.uo aggress.uo nim.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. To use MLton, you list all your modules in an MLB file[21], and MLton compiles them at one go. If you want to try this, download files `sticks.mlb`[22] and `runsticks.sml`[23], and then compile with compile with

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

[21] http://mlton.org/MLBasisSyntaxAndSemantics
[22] ./sticks.mlb
[23] ./runsticks.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 `mlton.org`[24].

Note: At press time, `Unit` module wasn't working with MLton.

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[25] • 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[26] • 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[27] • Code is almost twice as large as necessary to solve the problem. • *Or*, code contains near-duplicate functions (most likely in AGS)

Performance and correctness

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

[24] http://www.mlton.org/
[25] http://www.cs.tufts.edu/comp/105/handouts/redundant-ml-cases.html
[26] http://www.cs.tufts.edu/comp/105/handouts/redundant-ml-cases.html
[27] http://www.cs.tufts.edu/comp/105/handouts/redundant-ml-cases.html

	Exemplary	**Satisfactory**	**Must Improve**
Correctness	• AGS code makes no additional assumptions about the implementations of `Player`, `Move`, or `Game`.		• AGS code assumes that players take turns.
Performance	• The AGS implements its `bestmove` and `forecast` functions using a single, pruned search that stops once the best move or outcome is known. • *Or*, the AGS implements `bestmove` and `forecast` by making just *one* search through the state space of possible game configurations.	• Function `bestmove` or forecast may search the state space of possible configurations more than once.	• Function `bestmove` or forecast may search the state space of possible configurations more than twice.