Modules and Abstract Types

COMP 105 Assignment

Due Tuesday, April 9, 2019 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 supported by concepts and mechanisms that you'll learn in the context of Standard ML, which, not coincidentally, has one of the most expressive and powerful system-level abstraction mechanisms ever created.

In this assignment, you will

- Learn a bit about abstract data types
- 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
- Learn about system design by extending an existing system with new modules

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 exercise N, you implement a full set of operations on natural numbers of arbitrary precision. Natural numbers provide an ideal setting in which to explore the freedom of choice offered to you by abstract types. You have already implemented addition, subtraction, and some conversions, and you can build on your own code or on my solutions.

  In exercise I, you use an *unknown* implementation of natural numbers to implement *signed* integers. These so-called "bignums" are an important programming-language abstraction.
- In exercise A, you implement an unsophisticated (yet unbeatable) computer opponent that can be used to play two-player games. To help you understand and test this popular exercise, you will also implement exercise G: one of three two-player games. The computer opponent requires careful thought but not a lot of code: in essence, it boils down to one carefully crafted recursive function. The games offer a range of experience, but most require less thought and more code.
- In exercise ADT, you'll collect the *abstraction functions* and *representation invariants* from your implementations of natural numbers, signed integers, and games.

Overall, you'll answer reading-comprehension questions, you'll deliver four modules, and you'll assemble the abstraction functions and the invariants from your modules. As usual, you do reading-comprehension

questions on your own. You may tackle the modules either 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, students report spending the same amount of time on this assignment as on other assignments.

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 binaries. 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
```

If you run into any surprises, consult Appendix I below, which explains, in detail, 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`, `#quotient`, `#remainder`, `#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]
- There is a dire warning below about large integer literals. Short version: don't use them.

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

Reading

For this homework, you will need to understand the ML Modules[2] section of "Learning Standard ML[3]." The book chapter on modules, chapter 9, is not included in your edition—the chapter is in the middle of a major revision, and the current state of the draft is very confusing. Instead of the confusing draft, we recommend the sixth lesson on program design[4]: "Program design with abstract data types."

You'll also need to understand section 9.10.2, "Arbitrary-precision integer arithmetic", starting on page 729 of *Programming Languages: Build, Prove, and Compare*. This section is included in your abridged edition, even though the complete text of chapter 9 is not. The book section contains everything you *need* to know, but we recommend an additional handout, "Mastering Multiprecision Arithmetic"[5], which provides many hints about implementation.

Reading comprehension

These questions will help guide you through the reading. We recommend that you complete them before starting the other exercises below. You can download the questions[6].

1. (NOT ON THE READING.) Throughout the term, your code's functional correctness has been assessed by automated testing. The automated test scripts are intended not only to assign a grade but to identify the most important fault in the code. When our scripts find faults, how often do you understand the reports?

   Please answer "Always," "Mostly", "Sometimes", "Seldom", "Rarely", "Never", or "I don't read those reports":

2. Using one of the sources in the ML learning guide[7], 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[8] 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
```

[2] https://www.cs.tufts.edu/comp/105/readings/ml.html#ml-modules
[3] https://www.cs.tufts.edu/comp/105/readings/ml.html
[4] ../design/lessons.pdf
[5] ../readings/arithmetic.pdf
[6] ./cqs.sml.txt
[7] ../readings/ml.html
[8] ml.html

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 structure match the signature? That is, if we write

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

does the resulting code typecheck? Please answer yes or no.

(b) Does the signature expose enough information for us to write the following function? Please answer yes or no.

```
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? Please answer yes or no.

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

(d) If in part (b) or part (c), if it is not possible to write the function given, change the signature to make it possible. If necessary, please copy, paste, and edit your new version in here:

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

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

The new structure `Env'`, sealed with signature `ENV'`, is useless. Please explain *why* it is useless:

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 a game (exercise G).

3. An ML *functor* is a function that operates on the module level. Think of it as a "module in waiting" or a "module builder." A functor's *formal* parameters, if any, are specified by a *sequence of declarations*, and its *actual* parameters are given by a *sequence of definitions*. A functor's *result* is a structure. Read about functors in Harper, as recommended in the ML learning guide, then answer the questions below.

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. My "test set" functor needs access to these operations:

- A function that, given a test, returns a string identifying the student who wrote the test
- 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 declarations here ...
                 )
    :> TEST_SET = struct ... end  (* ignore this part *)
```

The formal parameters must include a declaration that specifies the type of a test, plus enough operations to provide the information needed above.

(b) Now focus your attention on one particular test, the `check-type` test. Its representation given by these definitions:

```
type uid = string
type check_type_test =
  uid * int * exp * ty  (* int is sequence number *)
```

The actual parameters to `TestSetFun` must give `check_type_test` as the type of test, and they must provide the operations specified by the formal parameters. 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 definitions here ...
          )
```

The important part here is knowing what definitions to write as actual parameters. The actual parameters must define all the types and the operations expected as formal parameters. You may also include as many *extra* definitions as you like—extra definitions are ignored. Here are some useful extra definitions:

```
fun uid          (u, _, _, _) = u
fun serialNumber (_, k, _, _) = k
fun exp          (_, _, e, _) = e
fun ty           (_, _, _, t) = t
```

When writing your the required definitions, feel free to use these code snippets:

- For comparison,

  ```
  case String.compare (uid1, uid2)
  ```

```
of EQUAL => Int.compare (seqno1, seqno2)
 | diff  => diff
```

- For string conversion,

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

  Assume that functions `expString` and `tyString` are given.

Please write your answer above where it says to fill in the definitions.

You now understand functors well enough to use them in exercises I and A.

4. Read about "signature refinement or specialization" in the ML learning guide[9]. 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 know how to refine the result signature of your Abstract Game Solver in exercise A.

5. Read about abstraction functions and invariants in the lesson "Program design with abstract data types"[10]. Then, from the ML homework, review the algebraic data type from the natural-number problems[11], and review the list-with-indicator[12] abstraction.

   Now answer these questions:

   (a) The lesson describes a sorted list as one possible representation for a set. Define a function `invariant` that takes as argument a list of integers and returns a Boolean saying if the list is sorted in strictly ascending order (that is, increasing, with no repeats). You may use ML or μScheme, and you may reuse any function assigned for homework this term:

   (b) In the ML homework, the algebraic type `nat` satisfies two invariants. In ML, define a function `invariant` of type `nat -> bool`, which returns true if and only if the given representation satisfies both invariants:

   (c) In the ML homework, the `'a ilist` represents an abstraction "list with indicator." I ask you to pretend that this abstraction is a value of type `'a indicated list`, where exactly one element is indicated, and `indicated` is defined by

   ```
   datatype 'a indicated
     = INDICATED   of 'a
     | UNINDICATED of 'a
   ```

   Using your chosen representation of `'a ilist`, or, if you did not complete the problem, the representation from the model solutions, define an ML function `absfun` of type `'a ilist -> 'a indicated list`, which acts as the abstraction function for the list with indicator. This function acts as the $\mathcal{A}$ function from the design lesson:

You are now ready to write abstraction functions and invariants in exercises I, N, C, and ADT.

[9] ../readings/ml.pdf
[10] ../design/lessons.pdf
[11] ./ml.html#nat
[12] ./ml.html#indicator

6. Read about short division starting on page 734 of the book, and in "Mastering Multiprecision Arithmetic"[13].

   (a) Divide 2918 by 7, calculating both quotient and remainder.
   At each step, you divide a two-digit number by 7. The remainder is passed along to form the next two-digit number.

   ```
    ________
   7 | 2 9 1 8
   ```

   At each step of the computation, you will take a two-digit dividend, divide by 7, and give quotient and remainder. The first step is

   ```
   02 divided by 7  ==  0 remainder 2
   29 divided by 7  ==  ...
   ```

   There are four steps in total. Edit the text above to state the dividend, divisor, quotient, and remainder at each step. Here, write the final four-digit quotient and the one-digit remainder:

   You are now ready to implement short division on natural numbers (for exercise N).

7. Going back to the same reading, and following the examples in the section "Using short division for base conversion," convert a number from decimal to binary and another number from decimal to octal.

   (a) Using repeated division by 2, convert decimal 13 to binary. The "Mastering Multiprecision Arithmetic"[14] handout shows the form, so please just fill in the right-hand sides here:

   ```
   q0 =          r0 =
   q1 =          r1 =
   q2 =          r2 =
   q3 =          r3 =
   ```

   Now write the converted numeral here:

   (b) Using repeated division by 8, convert decimal 25 to octal 31. Follow the same model: at each step, give the intermediate quotient and remainder, and then form the final quotient by reading off the remainders.

   You are now ready to implement the `decimal` operation on natural numbers (for exercise N). This will also enable you to implement the `toString` operation on signed integers.

Programming, part one: Arbitrary-precision numbers

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 the next two exercises, you will implement a true integer abstraction, called `bigint`, which *never* raises `Overflow` (but it could run out of memory).

You will implement this interface:

[13] ../readings/arithmetic.pdf
[14] ../readings/arithmetic.pdf

```
signature BIGNUM = sig
   type bigint

   exception BadDivision               (* contract violation for sdiv *)

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

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

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

        Given a d out of range or a negative n,
        sdiv (n, d) raises BadDivision

        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.
     *)

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 out of acceptable range *)

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

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

       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 exercises **I** and **N** below, you implement both interfaces. You can do them in either order.

Programming exercise I: Integers from natural numbers

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. (Variables will do nicely.)

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:

- Write the abstraction function in a comment. We recommend algebraic laws, with right-hand sides that 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 call it and 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 and one dire warning

The only major pitfall in this abstraction is that 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 me 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 a small-minded person hands you the most negative integer (see `Int.minInt`), its magnitude cannot be represented as a machine integer. To compute the magnitude of a negative integer safely, use the following steps:

1. Add one to the negative integer
2. Negate the sum
3. Convert the result to `nat`
4. Add one to the `nat`

The **dire warning** is this: do not include large integer literals, like `~4611686018427387904`, in your source code. Your code won't compile on our test machine, and you will get **No Credit**. You do not need to mess around with `Int.minInt`, but if you feel compelled to do so, refer to it by name.

Guidance: 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. Here's all you have to write:

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

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 result to the minuend:[15]

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

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

- When "signed zeroes" are involved, comparison becomes tricky. When I call `compare`, I 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
```

Related reading: Using the information in the ML learning guide[16], read about ML signatures, structures, and functors.

Programming exercise N: Arbitrary-precision natural numbers

In this exercise, you finish the implementation of natural numbers—of arbitrary precision—that you started on the first ML assignment[17]. Arithmetic will *never* overflow; the worst that can happen is you run out of memory. You will design and build an implementation of signature `NATURAL`, which you will

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

[16] ../readings/ml.pdf

[17] ml.html#arithmetic-by-pattern-matching-on-lists

put in file `natural.sml`. You will tackle the implementation in two parts: first choose a representation and invariant, then implement the operations.

ML syntax hint: The introduction and elimination forms for the quotient/remainder record are shown above.

How big is it? I wrote two implementations using two significantly different representations. Together they total about 270 lines of code, of which 50 lines are blank. The two implementations are roughly the same size.

What's the point? 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, the firewall is emplaced through opaque signature ascription, known to a select group of 105 alumni as "the beak."

Getting started: Representation, abstraction function, and invariant. A natural number should be represented by a sequence of digits. But "sequence of digits" has many representations! You probably want a list of digits, an array of digits, or an algebraic data type like the one from the ML homework[18]. And what counts as a "digit" depends on the base. In this assignment, the choice of base is yours, but to get full credit, you must choose a base that is different from 10.

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

- Write the abstraction function in a comment. We recommend algebraic laws, with right-hand sides that use standard arithmetic notation for operations on natural numbers or mathematical integers.
- Write the invariant in an ML function named `invariant` of type `nat -> bool`, located *inside* the module. Because it is located inside the module, the `invariant` function has complete access to the representation of `nat`.

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

  We recommend, but do not require, that the invariant rule out any representation with leading zeroes.

Put the abstraction function and invariant *inside* your `Natural` module, right after the definition of type `nat`.

Related reading:

- Read about "Choice of representation" on the first page of the handout "Mastering Multiprecision Arithmetic."[19]
- Read the short statement about *representation invariants* and *abstraction functions* under "Reading" in this homework.
- If you need a refresher on ML signatures, structures, and functors, there's the ML learning guide.[20]

Finishing up: Implementing the interface. In file `natural.sml`, define a structure `Natural` that implements signature `NATURAL`. The right way to do it is with code that looks like this:

[18] ml.html
[19] ../readings/arithmetic.pdf
[20] ../readings/ml.pdf

```
(* inconvenient code *)
structure Natural :> NATURAL = struct
  ...
  type     nat = ... your definition from part (a) ...,  OR
  datatype nat = ... a datatype is also OK here ...
  (* abstraction function:
       ... *)
  fun invariant ... = ...

  ...
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 = ...     OR
  datatype nat = ...
  (* abstraction function:
       ... *)
  fun invariant ... = ...

  ...
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.

Feel free to draw on solutions from the first ML assignment—just acknowledge your sources.

Related reading:

- For the details of addition, subtraction, and conversion, revisit the ML assignment[21]. For multiplication, look at the model solutions.
- Also from that assignment, revisit the "smart constructor"[22].
- If you want to explore another representation, the short handout "Mastering Multiprecision Arithmetic"[23] recommends exactly how to implement natural numbers, including what helper functions you will find most useful and what operations to implement first. Its recommendations are somewhat different from what you saw in the ML assignment, but if you want to use mutable arrays, you will find the recommendations useful.

[21] ml.html
[22] ml.html#smartcons
[23] ../readings/arithmetic.pdf

- Read about arithmetic in section 9.10.2 of *Programming Languages: Build, Prove, and Compare*.
- If you need a refresher on ML signatures, structures, and functors, consult the ML learning guide.[24]

Guidance for choosing a representation of natural numbers

No one representation is equally good for all operations.

- Addition and subtraction work from least-significant digit to most-significant digit. These operations work well with a little-endian list or with an array.
- Short division works from most-significant digit to least-significant digit. This operation can work well with any representation.
- Multiplication can proceed in either direction, and it can work well with any representation.

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.

If you'd like to use mutable arrays to implement natural numbers, here are some hints based on my experience:

- I recommend the invariant that every element d lies in the range $0 \leq d < b$. If the array contains $n + 1$ digits, the abstraction function maps it to the natural number $\sum_{i=0}^{n} d_i \cdot b^i$.
- To help you implement addition $X + Y$ and subtraction $X - Y$, I recommend defining an internal function `digit` that will help you pretend that X and Y are the same size: asking for a digit beyond the bounds of the array should return 0.

As you complete exercise N, you may want to revisit your representation choices. That's part of the point—representations can change without affecting any external code.

Guidance for implementing natural-number operations

Choice of representation determines what's hard and what's easy. Here are a few notes and hints:

- The first ML assignment includes exercises on addition and subtraction of little-endian lists of decimal digits, where the list is represented using a special-purpose algebraic data type. (Multiplication was extra credit.) These codes can readily be adapted to bases much larger than 10. You are welcome to use your own code or the model-solution code—just *acknowledge your sources*.
- Comparison can be implemented directly on sequences of digits, or you can take the easy way out and use subtraction: `n < m` if and only if `n /-/ m` raises `Negative`. And `n > m` if any only if `m < n`. And `m = n` if and only if neither `n < m` nor `m < n`.

[24] ../readings/ml.pdf

- The algebraic laws in the book specify operations on numbers of the form $n \cdot b + d$. These are a natural fit for an algebraic data type.
- As with your implementation of `BIGNUM`, I encourage you to define infix operators internally:

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

- Here's a **mistake to avoid:** Don't 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.

Testing numbers

Testing should begin with natural numbers. To test anything, you need `ofInt` 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 parts 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, using only addition:

```
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.

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

To create more test cases, you can use the interactive `ghci` interpreter on the servers. It implements the functional language Haskell, in which large-integer arithmetic is the (sensible) default. Here's an example:

```
$ ghci
GHCi, version 8.0.2: http://www.haskell.org/ghc/  :? for help
Prelude> 1234567890 * 987654321
1219326311126352690
Prelude> 1234567890 * 987654321 * 9999999999
```

```
121932631100442005888736473l0
Prelude>
```

To use such big numbers in your own code, you can convert them from strings. Here is a rough sketch of conversion from string to `nat`:

```
fun digitOfChar c =
  if Char.isDigit c then Char.ord c - Char.ord #"0"
  else raise Match

val natOfDigit = Natural.ofInt o digitOfChar

val ten  = Natural.ofInt 10
val zero = Natural.ofInt 0

val /*/ = Natural./*/
val /+/ = Natural./+/

infix 7 /*/
infix 6 /+/

fun natOfString s =
  foldl (fn (d, n) => n /*/ ten /+/ natOfDigit d) zero (explode s)
```

Beyond these simple kinds of tests, skilled engineers 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 provide some computer-generated tests for natural numbers[25]. To run the tests, compile everything using `compile105-sml`, then load them into the interactive system, apply the functor, and run the tests:

```
$ mosml -P full -I /comp/105/lib
Moscow ML version 2.10-3 (Tufts University, April 2012)
Enter `quit();' to quit.
- load "natural-tests";
> val it = () : unit
- load "natural";
> val it = () : unit
- structure Run = UnitTestsFun(structure Natural = Natural);
> structure Run : {val run : unit -> unit}
- Run.run();
All 72 internal Unit tests passed.
> val it = () : unit
```

I also provide some generated tests for signed integers[26], which require something like the following:

```
- structure B  = BignumFn(structure N = Natural);
```

[25] ./natural-tests.sml
[26] ./bignum-tests.sml

```
- structure BT = UnitTestsFun(structure Bignum = B);
- BT.run();
```

Programming, part two: Two-player games of complete information

Abstraction supports reuse. In this part of the assignment, you'll reuse code for an idea at a very abstract level: "play a game." In particular,

- You will work with an interface that defines what it means to be a playable game. This interface mixes abstract data types with "manifest" data types (exposed representations).
- You will create an implementation of the game interface. From the games on the games page[27], you'll choose the one you most wish to implement. After you choose a game, you'll also choose representations of the game's key abstractions.
- You will reuse my code to play your game against a computer opponent.
- You will build your own computer opponent, the Abstract Game Solver, which will be capable of playing any (sufficiently small, terminating) "game of complete information."

An abstraction of two-player games

To make it possible to reuse the *same* code for playing games, no matter what the game, requires a carefully designed abstraction. That abstraction, which is based on a design by George Necula[28], is presented here.

As always, program design begins with data. The data in this exercise 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.

The abstraction views every game as 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 interface to the abstraction 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*. 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

  val otherplayer : player -> player
  val unparse     : player -> string
```

[27] games.html

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

```
  val outcomeString : 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 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: Moves

The move abstraction exists to help communicate gameplay to a human player. Human input is converted to a move by function parse, and functions prompt and visualize are used to request and show 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 states and moves)

The contract for an entire game is given in 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 central abstraction is `state`, as described above. A state includes complete information about a game in progress, including 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 that 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 is a broad interface. For example, there are three different ways to tell if a game is over. They must all agree!

Third abstraction: a game solver

The design of the `GAME` interface enables us to build a computer opponent based on exhaustive search: an abstract game solver (AGS). An AGS tries all possible moves, and if it can find a move that leads to a win, it chooses that move. The search 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 interface to an AGS is narrow: all we can do is present a state, and ask the solver what a player (whose turn it is) should do in that state. If the state is final, the AGS has no recommendation. Otherwise, the AGS recommends a legal move. Either way, the AGS says what final outcome is expected, assuming that its recommendation (if any) is followed and that no player ever makes a bad 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
  val advice : Game.state -> { recommendation : Game.Move.move option
                             , expectedOutcome : Player.outcome
                             }
    (* 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.

Design rationale

The abstractions above are designed around three needs:

- In order to have a computer opponent, we need to be able to build an AGS.
- In order to *play* against the computer opponent, we need a "play manager," which converts key information to and from strings, for the benefit of the human player.
- We need both the AGS and the play manager to work *unchanged* with a variety of simple games, such as Tic-Tac-Toe, Connect 4, and peg solitaire.

These needs explain the design of the interfaces:

- For moves, the AGS just needs a value. But for the human player, we need to convert moves to and from strings, thus we have exported functions `prompt`, `visualize`, and `parse`.
- For states, all clients need an initial state, and they need to produce new states by making moves (exported functions `initial` and `makemove`). Similarly, both the play manager and the AGS need to know whose turn it is (`whoseturn`) and when a state is final (`isOver`), as well as the `outcome` of a final state. But only the play manager needs `visualize`, and only the AGS needs to enumerate all legal moves (`legalmoves`).

Programming exercise G: Implement a game

The main foci of this assignment are the large integers and the Abstract Game Solver. But to understand how the solver works, it helps you to implement a simple two-player game. You have your choice of 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.)

Each game is more interesting to play (and more time-consuming to implement) than the game before it. Depending on which game you choose, you will 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[29]. The instructions include the strings that must be recognized by `Move.parse`. **You need these instructions!**

I have implemented all three games. You can play them by running `coins`, `sticks`, or `ttt`, all of which can be found in `/comp/105/bin`. 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. The main thing we'll look for in testing is to be sure that 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.

[29] games.html

- 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 lengthy description of the `GAME` signature, the instructions for the game of your choice on the games page[30], and the section on ML modules in the ML learning guide[31].

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 a game-specific AGS, then use it with my play manager. 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, which uses *take the last coin* as an example:[32]

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

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

```
- load "play";           <---- my play manager
> val it = () : unit
- structure P = PlayFun(structure Ags = CAgs);        <--- 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.state -> Game.Move.move
    (* raises Quit if human player refuses to provide a move *)

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

[30] games.html

[31] ../readings/ml.pdf

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

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 play manager uses to request a move given a configuration. The idea is that the function returned asks either the AGS or the human for a move, depending on whose turn it is.

The function `play` expects an input function (one built by `getamove`) and an initial state. This function then starts an interactive loop which prints visualizations and prompts 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 = Coins.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 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:

```
- P.play computerxo cnfi;
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.

Programming exercise A: Build an Abstract Game Solver

Implement an Abstract Game Solver. 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 even 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 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**. 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*. Here is what I recommend:

- 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.[33]
- Depending on how you choose to tackle the exercise, you may have to compare two outcomes to see which one is better (relative to a particular player). *Avoid breaking down this comparison into nine cases.* Instead, try converting each outcome to some kind of numeric score, and identify the better outcome with a higher score.

Write an AGS using 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
```

Note how annoying the `where type` declarations are: they look tautological, but they're not. Complain to Dave MacQueen[34] and Bob Harper[35].

Hints:

- *ML syntax:* The introduction and elimination forms for the `advice` record are analogous to the forms for the quotient/remainder record described above. Only the names of the fields are different.
- **Use the design process.** In particular, remember the structure of a recursive function that consumes a nonempty list of moves.
- You might be tempted to handle the non-final case by using `map` with legal moves. If you do, your AGS will always search *every* possible move, even if it finds a winning strategy on the very first move. Such code will make your AGS slow and no fun to play. Instead of using `map`, write a recursive function that cuts off the search once a winning move is found. 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 the game you have implemented. Use `whoseturn` and `outcome`

[33] 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`."

[34] http://people.cs.uchicago.edu/~dbm/

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

instead. We will test your AGS on games that are quite different, including 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 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, 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 coin game. If you have implemented one of the simple games, and 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[36].
- The model solutions for nonempty lists on the `scheme` homework[37], including the model solution for `arg-max`[38]

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 `legalmoves`). Writing the code would become very challenging—polymorphism and higher-order programming at the value level is the 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 a language construct 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 |   |
-------------
|   |   |   |
```

[36] ../readings/ml.pdf
[37] scheme.html#nonempty_lists
[38] scheme.html#argmax

```
-------------
```

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 : ...
- load "play";
> val it = () : unit
- structure PT = PlayFun(structure Ags = TTTAgs);
> structure PT : ...
- PT.play (PT.getamove [Player.O]) (TTT.initial Player.X);
-------------
|   |   |   |
-------------
|   |   |   |
-------------
|   |   |   |
-------------
Player X is to move

Square for player X?
```

Our Tic-Tac-Toe 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.

Proof. Here's a conjecture: Every state in any correct implementation of the `GAME` interface always satisfies exactly one of these three properties:

- 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.

If the conjecture is true, prove it. If not, provide a counterexample.

If you find a counterexample, can the conjecture be repaired? If so, repair it, and prove the repaired conjecture. If not, explain why not.

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.

Programming wrapup: Abstraction functions and invariants

Exercise ADT. *Collect representations, abstraction functions, and invariants.*

In a new file `adt.txt`, summarize your design work:

- Copy and paste your definitions of `bigint`, `nat`, `move`, and `state`, along with the associated abstraction functions and invariants. If you implemented any extra-credit games, include those definitions of `state` and `move` also.

  If you're not sure how to write the abstraction functions, look at the examples in the 6th lesson on program design.

- Separate each group with a long line of dashes, thus:

  ```
  ----------------------------------------------------------------
  ```

- Below each group, explain in one or two sentences *why* you chose the representation that you did. The choices you explain should include both the type definitions and other important choices like the base of natural numbers.

  All defensible reasons for choosing a representation are legitimate and will earn full marks. Examples of defensible reasons include the following:

  - "We wanted to keep the invariant as simple as possible."
  - "We wanted to make it easy to find the legal moves."
  - "We wanted to make it very fast to detect three in a row."
  - "We wanted to see if we could make it work with base 2."

  Please avoid *indefensible* reasons like, "it seemed like a good idea at the time." Say *why* it seemed like a good idea.

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 exercises, submit 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 Proof or Game Theory extra credit, if you choose to do them
- File `bignum.sml`, implementing the functor which builds signed integers on top of natural numbers (your solution to exercise I)
- File `natural.sml`, implementing your solutions to exercise N
- 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
- For exercises G and A, your `coins.mlb`, `sticks.mlb`, or `ttt.mlb` file. (See Appendix I for information on the this file.)
- For exercises G and A, any other files you need in order to compile your game and your AGS
- For exercise ADT, file `adt.txt` with your choices, reasons, abstraction functions, and invariants.

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 `compile105-sml` 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[39]. George Necula[40], 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. (MLton is going to be important when the time comes to test your game.)

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 `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.

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 coins.sml
```

This compilation produces two files:

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

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

- When you are debugging, you'll want to load compiled modules into the interactive system. Load them directly using `load`, e.g.,

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

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

```
: homework: mosml -I /comp/105/lib -P full
Moscow ML version 2.10-3 (Tufts University, April 2012)
Enter `quit();' to quit.
- load "coins";
> 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. To do anything interesting, one of the source files should have a top-level call to `play`, `advice`, 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 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. To use MLton, you list all your modules in an MLB file[41], and MLton compiles them at one go. If you want to try this with the coins game, download files `coins.mlb`[42] and `runcoins.sml`[43], and then compile with compile with

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

(You can also download and compile `sticks.mlb`[44], `runsticks.sml`[45], `ttt.mlb`[46], and `runttt.sml`[47].)

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`[48].

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.

[41] http://mlton.org/MLBasisSyntaxAndSemantics
[42] ./coins.mlb
[43] ./runcoins.sml
[44] ./sticks.mlb
[45] ./runsticks.sml
[46] ./ttt.mlb
[47] ./runttt.sml
[48] http://www.mlton.org/

	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[49] • 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[50] • 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[51] • 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.

	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.

	Exemplary	**Satisfactory**	**Must Improve**
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. • Long computations on natural numbers are basically instantaneous, even when hundreds of decimal digits are involved. • *Or*, we can do hundreds of operations on natural numbers, including multipliation, on numbers with dozens of decimal digits, in less then a hundred milliseconds.	• Function `advice` may search the state space of possible configurations more than once. • The natural-number tests I provide run in less than a hundred milliseconds.	• Function `advice` may search the state space of possible configurations more than twice. • Any computation on natural numbers takes more than a second.