Modules and Abstract Types

COMP 105 Assignment

Due Wednesday, November 14, 2018 at 11:59PM

Contents

Overview 1

Setup 2

DIRE WARNING 2

Reading 2
  Representation invariants ...................................................... 2
  Arithmetic ................................................................................. 3

Reading comprehension 3

Programming, part one: Arbitrary-precision numbers 6
  Programming problem I: Integers from natural numbers ................... 8
    Choice of representation ....................................................... 9
    Two mild warnings and one dire warning ................................ 9
    Guidance: infix operators ....................................................... 10
    Guidance: algebraic laws ....................................................... 10
  Programming problem N: Arbitrary-precision natural numbers .......... 11
    Guidance for choosing a representation of natural numbers .......... 14
    Guidance for implementing natural-number operations ............... 14
  Testing numbers ........................................................................ 15

Programming, part two: Two-player games of complete information 17
  Understanding and representing adversary games .......................... 17
  The idea behind the Abstract Game Solver (AGS) ......................... 17
  Basic data in the problem: Players and outcomes ......................... 18
  Specification of an abstract game: the GAME signature ................ 19
  Specification of a game solver: the AGS signature ....................... 20
  Programming problem S: Implement “take the last stick” ............... 21
  Integration testing, part I: your game with my AGS ..................... 23
  Programming problem A: Build an Abstract Game Solver ................ 25
    A common mistake to avoid when debugging your AGS ............... 27
  Integration testing, part II: your AGS with my game .................... 27
  Descriptions of two-player games ............................................. 28
Overview

In this assignment, you will

- Learn a bit about abstract data types
- Revisit arbitrary-precision arithmetic, which you will see yet again next time
- 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 three 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 problem 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 problem I, you use an unknown implementation of natural numbers to implement signed integers. These so-called “bignums” are an important programming-language abstraction.

- 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.

Overall, you’ll answer reading-comprehension questions and you’ll deliver four modules. As usual, you do reading-comprehension questions on your own. You may tackle the modules 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:

```ml
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

```ml
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 WARNING**

DON’T USE `#quotient` and `#remainder`. USE PATTERN MATCHING INSTEAD.

**Reading**

For this homework, you will need to understand the ML Modules\(^1\) section of “Learning Standard ML\(^2\).” 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 provide a few excerpts and hints here.

**Representation invariants**

A *representation invariant* tells us what is true about the representations we encounter at run time. It could be something as simple as “the list contains no duplicate elements” (for the implementation of sets as lists) or something so complicated as to demand to be written mathematically. Interesting data structures usually satisfy multiple representation invariants. These may be referred to individually, and they may also be collectively called “the invariant.”

A good example is a binary search tree. A binary search tree always has an *order invariant*—usually “smaller to the left, larger to the right.” The order invariant guarantees that a search function will find

---

\(^1\)https://www.cs.tufts.edu/comp/105/readings/ml.html#ml-modules

\(^2\)https://www.cs.tufts.edu/comp/105/readings/ml.html
an object, if present, without having to look at every node of the tree. A serious, sophisticated binary search tree also has a balance invariant. There are many different forms of balance invariant, but they all guarantee that search takes a number of steps that is at most logarithmic in the number of nodes in the tree.

Arithmetic

“Mastering Multiprecision Arithmetic” is a short handout plus an excerpt from the previous edition of the book.

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. Using one of the sources in the ML learning guide, 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 and from code you may have seen in the course interpreters:

```plaintext
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:

```plaintext
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

```ml
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.

```ml
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.

```ml
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, 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

```ml
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 the “pick up the last stick” game (problem S).

2. 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. 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:

```ml
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:

```ml
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:

```ml
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:

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

- For string conversion,
  ```ml
  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 problems I and A.
3. Read about “signature refinement or specialization” in the ML learning guide\textsuperscript{7}. Now, 
   (a) Explain what, in part (b) of the previous question, is going on with the \texttt{where} type syntax. 
   (b) Explain what would go wrong if we wrote this code instead:

   \begin{verbatim}
   structure CheckTypeSet :> TEST_SET = TestSetFun(...)
   \end{verbatim}

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

4. In “Mastering Multiprecision Arithmetic”\textsuperscript{8}, read the section on short division, plus pages 
   606 and 607 of the excerpt appended to that handout.  
   (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.

   \begin{verbatim}
   \begin{tabular}{cc}
   7 & 2 9 1 8 \\
   \end{tabular}
   \end{verbatim}

   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

   \begin{verbatim}
   0 2 \text{ divided by } 7 == 0 \text{ remainder } 2
   2 9 \text{ divided by } 7 == ...
   \end{verbatim}

   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 problem N).

5. 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” handout shows the form, so please just fill in the right-hand sides here:

   \begin{verbatim}
   q0 = r0 =
   q1 = r1 =
   q2 = r2 =
   q3 = r3 =
   \end{verbatim}

   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 \texttt{decimal} operation on natural numbers (for problem N). 
This will also enable you to implement the \texttt{toString} operation on signed integers.

---
\textsuperscript{7}../readings/ml.pdf
\textsuperscript{8}../readings/arithmetic.pdf
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 problems, you will implement a true integer abstraction, called bigint, which never overflows (but it could run out of memory).

You will implement this interface:

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 <= base 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

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

  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 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 larger than base *)

  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 <= base,
     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

     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 (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 problems I and N below, you implement both interfaces. You can do them in either order.
Programming problem 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:

```sml
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).

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

Choice of representation

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.

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. The cheap and cheerful way to compute the magnitude is

(a) Add one to the negative number
(b) Negate the sum
(c) Convert the result to nat
(d) 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:

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

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

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

Infix 6 /+/ /-/
infix 7 /*/

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

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

- **Addition:**
  ```plaintext
  +N <+> +M == +(N /+/ M)
  +N <+> -M == +(N /-/ M) == -(M /-/ N)
  -N <+> +M == -(N /-/ M) == +(M /-/ N)
  -N <+> -M == -(N /+/ M)
  ```
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.\(^9\)

\[ +N \leftrightarrow +M = +N \leftrightarrow -M \]

... 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:

\[ \text{compare } (+0, -0) = \text{EQUAL} \]

Here are some more general laws:

\[ \text{compare } (+N, +M) = N\text{.compare } (N, M) \]
\[ \text{compare } (-N, -M) = N\text{.compare } (M, N) \quad (* \text{ order is swapped } *) \]
\[ \text{compare } (-N, +M) = \text{LESS}, \quad \text{provided } N \text{ is not 0 or } M \text{ is not 0} \]
\[ \text{compare } (+N, -M) = \text{GREATER}, \quad \text{provided } N \text{ is not 0 or } M \text{ is not 0} \]

Related reading: Using the information in the ML learning guide\(^{10}\), read about ML signatures, structures, and functors.

Programming problem N: Arbitrary-precision natural numbers

In this problem, you finish the implementation of natural numbers—of arbitrary precision—that you started on the first ML assignment\(^{11}\). 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 put in file natural.sml. You will tackle the implementation in two parts: first choose a representation and invariant, then implement the operations.

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 and invariant. A natural number should be represented by a sequence of digits. But “sequence of digits” has many representations! To choose one, consider these questions:

- What is a digit? Or equivalently, what is the base $b$ used by your representation?

---

\(^9\)Words like “subtrahend” and “minuend” are useful, but I always have to look them up.

\(^{10}\)../readings/ml.pdf

\(^{11}\)ml.html#arithmetic-by-pattern-matching-on-lists
• 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 and short division, can leave leading zeroes in a representation. The number of leading zeroes must be controlled so that it doesn’t grow proportional to the number of operations used.

• Do you prefer a representation invariant that 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.

Before diving into the code, write two definitions:

• Using either type or datatype, define type nat, which determines your representation of natural numbers.
• Define function invariant of type nat -> bool, which says whether a given value of type nat satisfies your representation invariant.

Now proceed to the next part: embedding these definitions in an ML module (“structure”).

**Related reading:**

• Read about “Choice of representation” on the first page of the handout “Mastering Multiprecision Arithmetic.”
• 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.

**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:

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

---

12../readings/arithmetic.pdf
13../readings/ml.pdf
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 = ...
       (* 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.

**What to put in the implementation.** Your implementation should begin by defining your chosen representation and its invariant:

- Using either type or datatype, define a representation for type nat.
- Write comments explaining what representation you picked and why.
- Define the invariant function, of type nat -> bool. The invariant function is a predicate that examines any possible value of the representation and says whether the value is a “good” natural number. Here are a couple of “bad” representations for a big-endian list of base 100:

  \[
  \begin{align*}
  [0, 0, 7] & \quad (* \text{bad: leading zeroes} *) \\
  [2018] & \quad (* \text{bad: number in list is not a valid digit} *)
  \end{align*}
  \]

With that foundation in place, implement all the other operations of signature NATURAL. Again, feel free to draw on solutions from the first ML assignment—just acknowledge your sources.

**Related reading:**

- The ML assignment walks you through addition, subtraction, and conversion in detail.
- If you want to explore another representation, the short handout “Mastering Multiprecision Arithmetic”\(^{14}\) 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.
- Read the algebraic formulation of arithmetic and the associated examples in the excerpt from the prior edition of *Build, Prove, and Compare*, which is appended to “Mastering Multiprecision Arithmetic”\(^{15}\).
- For addition and subtraction, consult the excerpt appended to the handout, page 605; or Hanson, pages 305 to 307.

\(^{14}\)../readings/arithmetic.pdf
\(^{15}\)../readings/arithmetic.pdf
• For multiplication, consult the excerpt appended to the handout, page 606; or Hanson, pages 307 and 308.

• For short division, consult the excerpt appended to the handout, pages 606 and 607; or Hanson, pages 311 and 312 (the XP_quotient function). What Ramsey calls a “remainder” $r_i$ is called carry by Hanson.

• If you need a refresher on ML signatures, structures, and functors, consult the ML learning guide.16

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 works well with a big-endian list or an array.

• Multiplication can proceed in either direction, and 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

```ml
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 problem 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 problems on addition and subtraction of little-endian lists of decimal digits. (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 \texttt{Negative}. And \( n > m \) if any only if \( m < n \).
And \( m = n \) if and only if neither \( n < m \) nor \( m < n \).

• The algebraic laws for addition, subtraction, and multiplication work by operating on numbers of
the form \( n \cdot b + d \)—from the least significant digit on up. But short division works from the top
down, and I’m not aware of any nice algebraic laws that describe it.

• As with your implementation of \texttt{BIGNUM}, I encourage you to define infix operators internally:
\begin{verbatim}
infix 6 /+/
infix 7 /*/
\end{verbatim}

• Here’s a \textbf{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.

**Testing numbers**

Testing should begin with natural numbers. To test anything, you need \texttt{ofInt} and \texttt{decimal}, which probably also means \texttt{sdiv}. If you can’t get \texttt{sdiv} working and you’re desperate to write tests, try using base 10
and writing a special-purpose implementation of \texttt{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:
\begin{verbatim}
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
\end{verbatim}

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 \texttt{Negative} exception.

It is harder to test multiplication, but you can at least use repeated addition to test multiplication by \textit{small}
values.

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

To create more test cases, you can use the interactive \texttt{ghci} interpreter on the servers. It implements the
function 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
12193263110044200588873647310
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 nat0Digit = 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 */ nat0Digit d) zero (explode s)

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 provide some computer-generated tests for natural numbers\(^{17}\). 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";
  - load "natural-tests";
  - load "natural";
  - load "natural-tests-Fun(structure Natural = Natural)";
  - load "natural-tests"
  - Run.run();
  All 72 internal Unit tests passed.
> val it = () : unit

I may also be able to generate some tests for signed integers, but at press time, there are no such tests. I also provide some generated tests for signed integers\(^{18}\), which will require something like the following:

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

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

**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\(^{19}\) 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.

\(^{18}\)http://www.cs.berkeley.edu/~necula/

\(^{19}\)http://www.cs.berkeley.edu/~necula/
This method (“exhaustive search”) is suitable only for very small games. Nobody would use it for a game like chess, for example. Nevertheless, variations of this idea are used successfully even for chess; the idea is to stop or “prune” the search before it goes too far. 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

  (* Returns the player’s string representation *)
  val toString : player -> string

  (* Returns the outcome’s string representation *)
  val outcomeToString : outcome -> string
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.\textit{otherplayer} \ p, Player.X, Player.0, 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 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’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.
- 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

Note that N is a parameter, and may be any nonnegative value of type int. In other words, you don’t get
to pick N—we do.

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\(^\text{20}\) or tic-tac-toe, 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 (\(^\text{\hat{}}\)) 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.

\(^{20}\)https://en.wikipedia.org/wiki/Hexapawn
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\textsuperscript{21} signature from /comp/105/lib; the compile105-sml 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\textsuperscript{22}.

**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:

```mosml
: homework> mosml -I /comp/105/lib -P full
Moscow ML version 2.10-3 (Tufts University, April 2012)
Enter ‘quit();’ to quit.
- load "sticks"; "sticks"<---- your game
  val it = () : unit
- load "ags"; "ags"<---- my AGS
  val it = () : unit
```

\textsuperscript{21}/Unit.sig.txt

\textsuperscript{22}/readings/ml.pdf
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:

```ml
load "play";
```

```ml
val it = () : unit
```

```ml
structure P = PlayFun(structure Ags = S14Ags); <--- create the player
```

Functor PlayFun returns a structure that implements the following signature:

```ml
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.

```ml
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:

```ml
- 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:
Programming problem A: Build an Abstract Game Solver

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:

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

where `result` is a representation you choose. The idea is that

- 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\(^{23}\) and Bob Harper\(^{24}\).

**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 with 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\(^{25}\).

**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

\(^{23}\)http://people.cs.uchicago.edu/~dbm/
\(^{24}\)http://www.cs.cmu.edu/~rwh/
\(^{25}\)../readings/ml.pdf
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 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 : ...
- load "play";
> val it = () : unit
```

28
- 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.

**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 | 0 | |
-------------
| 0 | 0 | X |
-------------
| 0 | X | |
-------------
| 0 | X | O |
-------------
| X | O | |
-------------
| X | 0 | |
-------------
| X | X | O |
-------------
| X | O | |
-------------
| X | 0 | |
-------------
| 0 | X | O |
-------------
| 0 | O | X |
-------------
```

29
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.

| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |

-----------

30
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)
  - Answers to Proof and Game Theory extra credits.
- File `bignum.sml`, implementing the functor which builds signed integers on top of natural numbers (your solution to problem I)
- File `natural.sml`, explaining which representation you picked for natural numbers, and why, and 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 sticks.mlb file. (See Appendix 1 for information on the this file.)
• 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 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\textsuperscript{26}. George Necula\textsuperscript{27}, 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 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
```

\textsuperscript{26}http://www.cs.cmu.edu/~rwh/
\textsuperscript{27}http://www.cs.berkeley.edu/~necula/
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 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, you’ll want to load 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 = ()
- structure Sticks14 = SticksFun(struct val N = 14 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\(^{28}\), and MLton compiles them at one go. If you want to try this, download files sticks.mlb\(^{29}\) and runsticks.sml\(^{30}\), and then compile with compile with

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

\(^{28}\)http://mlton.org/MLBasisSyntaxAndSemantics
\(^{29}\)./sticks.mlb
\(^{30}\)./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.\(^{31}\)

### 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.

<table>
<thead>
<tr>
<th>Exemplary</th>
<th>Satisfactory</th>
<th>Must Improve</th>
</tr>
</thead>
<tbody>
<tr>
<td>All modules are sealed using the opaque sealing operator :&gt;</td>
<td>Most modules are sealed using the opaque sealing operator :&gt;</td>
<td>Only some or no modules are sealed using the opaque sealing operator :&gt;</td>
</tr>
<tr>
<td>Code uses basis functions effectively, especially higher-order functions on list and vector types.</td>
<td>Code uses the familiar functions, but misses opportunities to use unfamiliar functions like Vector.tabulate.</td>
<td>A module is defined without ascribing any signature to it (unsealed)</td>
</tr>
<tr>
<td>Code has no redundant case analysis(^{32})</td>
<td>Code has one redundant case analysis(^{33})</td>
<td>Code misses opportunities to use map, fold, or other familiar HOFs.</td>
</tr>
<tr>
<td>Code is no larger than is necessary to solve the problem.</td>
<td>Code is somewhat larger then necessary to solve the problem.</td>
<td>Code has more than one redundant case analysis(^{34})</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Code is almost twice as large as necessary to solve the problem.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Or, code contains near-duplicate functions (most likely in AGS)</td>
</tr>
</tbody>
</table>

#### 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.

<table>
<thead>
<tr>
<th>Exemplary</th>
<th>Satisfactory</th>
<th>Must Improve</th>
</tr>
</thead>
<tbody>
<tr>
<td>AGS code makes no additional assumptions about the implementations of Player, Move, or Game.</td>
<td></td>
<td>AGS code assumes that players take turns.</td>
</tr>
</tbody>
</table>

\(^{31}\)http://www.mlton.org/
<table>
<thead>
<tr>
<th>Exemplary</th>
<th>Satisfactory</th>
<th>Must Improve</th>
</tr>
</thead>
<tbody>
<tr>
<td>• The AGS implements its bestmove and forecast functions using a single, pruned search that stops once the best move or outcome is known.</td>
<td>• Function bestmove or forecast may search the state space of possible configurations more than once.</td>
<td>• Function bestmove or forecast may search the state space of possible configurations more than twice.</td>
</tr>
<tr>
<td>• Or, the AGS implements bestmove and forecast by making just one search through the state space of possible game configurations.</td>
<td>• The natural-number tests I provide run in less than a hundred milliseconds.</td>
<td>• Any computation on natural numbers takes more than a second.</td>
</tr>
<tr>
<td>• Long computations on natural numbers are basically instantaneous, even when hundreds of decimal digits are involved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Or, we can do hundreds of operations on natural numbers, including multiplication, on numbers with dozens of decimal digits, in less than a hundred milliseconds.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>