Core ML

CS 105 Assignment

Due Tuesday, October 25, 2022 at 11:59PM

Contents

Overview **1**

Prelude **1**

Setup . . . 1
ML's initial basis . . . 2
Unit testing . . . 2
Things to review before starting . . . 3

What we expect **4**

We expect the right types . . . 4
We expect wise, well-formatted unit tests . . . 5
We expect case analysis only when necessary . . . 5
We don't expect *written* algebraic laws . . . 6
We expect an acceptable style . . . 6
We expect you to remove redundant parentheses . . . 6
We expect you to avoid forbidden functions and constructs . . . 6

Reading comprehension **7**

Problems to solve individually **7**

Revisiting lists (30%) . . . 8
Revisiting S-expressions (10%) . . . 9
Arithmetic on digit sequences (40%) . . . 10

With a partner: Closures (20%) **15**

Extra credit **18**

Avoid common mistakes **18**

What to submit and how to submit it **19**

Submitting your individual work . . . 19
Submitting your improved μScheme interpreter . . . 19

How your work will be evaluated **20**

Overview

Types are everywhere. All languages that aspire to reliability at scale have type systems, and one of the programming-language stories of the 2020s has been the migration of popular languages to better, more effective type systems (JavaScript to TypeScript, PHP to Hack, and C to Rust).

This assignment gets you started programming with types, and also with *pattern matching*. Pattern matching allows most algebraic laws (just the algorithmic ones) to be executed directly as code—no more translation, and no more `null?`, `car`, or `cdr`!

The combination of types and pattern matching is found in such languages as Standard ML, Haskell, Elm, OCaml, Reason, F#, Scala, Idris, Agda, and Coq/Gallina. These languages excel at analysis and implementation of other languages; they are the technology of choice for applications like compilers, verification, and other static analysis, including of security properties. In the next few weeks, you will explore such applications by using Standard ML to implement type systems.

The assignment has two parts:

- As on previous assignments, you will write many small functions on your own.
- Possibly working with a partner, you will make a small change to the μScheme interpreter that is written in ML (in Chapter 5).

After completing the assignment, you will be ready to tackle serious programming tasks in Standard ML.

Prelude

Setup

CS 105 uses two different implementations of Standard ML. For the small problems, we recommend Moscow ML, which is in `/usr/sup/bin/mosml`. To start Moscow ML, use

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

If everything is working correctly, you should see this prompt:

```
Moscow ML version 2.10-4 (Tufts University, April 2017)
Enter `quit();' to quit.
-
```

If you don't see the Tufts name, send an immediate email of complaint to `staff@cs.tufts.edu`, with a copy to `nr@cs.tufts.edu`.

If you get a baffling error message, try compiling your code with `mlton` (pronounced "Milton"):

```
mlton-with-unit -verbose 1 -output a.out warmup.sml
```

MLton generates excellent native code and offers superior error messages. But it is can be slow to compile, so we recommend it only when performance is an issue or when you need a good error message.

Because of its superior performance, we do recommend `mlton` for the large problem.

ML's initial basis

As in the `hofs` and `continuations` assignments, we expect you to use the initial basis, which is properly known as the Standard ML Basis Library. By the standards of popular languages, the basis is small, but it is still much more than you can learn in a week. Fortunately, you only have to learn a few key parts:

- Type constructors `list`, `option`, `bool`, `int`, `string`, and `order`
- Modules `List` and `Option`, including `List.filter`, `List.exists`, `List.find`, and others
- Other module functions `Int.toString`, `Int.compare`, and `String.compare`
- Top-level functions `o`, `print` (for debugging), `map`, `app`, `foldr`, `foldl`
- Our own `Unit` module, which is not part of the Basis Library but is described in our guide *Learning Standard ML* (Unit Testing).

The most convenient guide to the basis is the Moscow ML help system; type

```
- help "";
```

at the `mosml` interactive prompt. The help file is badged incorrectly, but as far as I know, it is up to date.

If you have Jeff Ullman's book, you need to know that Chapter 9 describes the 1997 basis, which is out of date: today's compilers use the 2004 basis, which is a standard. But there are only a few differences, primarily in I/O and arrays. The most salient difference is in the interface to `TextIO.inputLine`, which you won't need to use in this course.

Unit testing

Regrettably, Standard ML does not have `check-expect` and friends built in. Unit tests can be simulated by using higher-order functions, but it's a pain. We provide such functions in a `Unit` module, which can be used to write these kinds of tests:

```
val () =
    Unit.checkExpectWith Int.toString "2 means the third"
    (fn () => List.nth ([1, 2, 3], 2))
    3

val () =      (* this test fails *)
    Unit.checkExpectWith Bool.toString "2 is false"
    (fn () => List.nth ([true, false, true], 2))
    false

val () = Unit.reportWhenFailures ()
```

If you include these tests in your `warmup.sml` file,[1] you can run them on the Unix shell command line, using `mosmlc` (with a "c"):

```
$ mosmlc -o a.out -I /comp/105/lib warmup.sml && ./a.out
In test '2 is false', expected value false but got true
One of two internal Unit tests passed.
$
```

[1]Using `Unit` tests at the interactive prompt is a little wacky. If you really want to do it, you can figure out how, but I'm going to discourage you—the mechanism you would need could creep into a `.sml` file, and if used there, it will create chaos and confusion. Just leave your `Unit` tests in files, where they belong.

You'll use `Unit.checkExpectWith` to write your own unit tests. You'll also use `Unit.checkAssert` and `Unit.checkExnWith`. The details are in *Learning Standard ML* in the section on Unit Testing. This section explains the unit-test interface and how create the string builders that `Unit.checkExpectWith` needs. Examples using common types might look like this:

```
val checkExpectInt        = Unit.checkExpectWith Unit.intString
val checkExpectIntList    = Unit.checkExpectWith (Unit.listString Unit.intString)
val checkExpectStringList = Unit.checkExpectWith (Unit.listString Unit.stringString)
val checkExpectISList =
  Unit.checkExpectWith (Unit.listString
                       (Unit.pairString Unit.intString Unit.stringString))
val checkExpectIntListList =
  Unit.checkExpectWith (Unit.listString (Unit.listString Unit.intString))
```

Running your unit tests

Let's look again at the example code above.

```
$ mosmlc -o a.out -I /comp/105/lib warmup.sml && ./a.out
In test '2 is false', expected value false but got true
One of two internal Unit tests passed.
$
```

There are two commands there:

- The `mosmlc` command *compiles* the code into executable binary `a.out`.
- The `./a.out` command runs the code (and the unit tests).

Students frequently compile their unit tests without running them. That can lead to unpleasant surprises. Remember the `./a.out`.

When I'm developing software, I often include a failing unit test on purpose. That makes it really obvious that the unit tests are actually being run. I don't remove the failing unit test until the code is ready to ship.

Things to review before starting

We provide a guide to *Learning Standard ML*. *Learning Standard ML* will guide you to other reading.

The fourth Lesson in Program Design explains how to apply our nine-step design process with types and pattern matching. This lesson includes the key code excerpts needed to design and program with standard type constructors `list`, `option` `bool`, `int`, `string`, and `order`, as well as the `exp` constructor from μScheme. Immediately following the lesson, you will find a one-page summary of ML syntax.

What we expect

We expect you will submit code that compiles, has the types given in the assignment, is acceptably styled, is tested, avoids redundant case analysis, and avoids forbidden functions and constructs. Code that does not compile, that has the wrong types, or that uses forbidden functions or constructs will earn **No Credit**. Code that is untested or has redundant case analysis may earn disappointing grades for structure and organization.

We expect the right types

As always, your code is assessed in part through automated testing. To be testable, each function must not only have the correct *name*; it must also have the correct *type*. Your type definitions must also match the type definitions given in the assignment.

To help you check types, we provide a script that analyzes your solution and reports on any type mismatches that it finds. On the command line, run

```
ml-function-check warmup.sml
```

The "function check" reports both missing and ill-typed functions. Any function that does not pass the function check will earn **No Credit**. The function check looks at the types listed here:

```
(* first declaration for function check *)
val mynull : 'a list -> bool
val reverse : 'a list -> 'a list
val minlist : int list -> int
exception Mismatch
val zip  : 'a list * 'b list -> ('a * 'b) list
val ziptoo : 'a list * 'b list -> ('a * 'b) list
val pairfoldrEq : ('a * 'b * 'c -> 'c) -> 'c -> 'a list * 'b list -> 'c
val concat : 'a list list -> 'a list
datatype sx
  = SYMBOL of string
  | NUMBER of int
  | BOOL   of bool
  | SXLIST of sx list
val numbersSx : int list -> sx
val flattenSyms : sx -> string list
datatype nat
  = ZERO
  | TIMES10PLUS of nat * int
val times10plus : nat * int -> nat
val intOfNat : nat -> int
val natOfInt : int -> nat
val natString : nat -> string
val carryIntoNat : nat * int -> nat
val addWithCarry : nat * nat * int -> nat
val addNats : nat * nat -> nat
exception Negative
val borrowFromNat : nat * int -> nat
val subWithBorrow : nat * nat * int -> nat
val subNats : nat * nat -> nat
val mulNats : nat * nat -> nat
(* last declaration for function check *)
```

We don't promise to have remembered all the functions and their types here. The `ml-function-check` script will help you, but making sure that every function has the right type is your job, not ours.

We expect wise, well-formatted unit tests

By this time, we expect that you understand the value of unit tests. Grading will focus on your code; except where specifically requested below (natural-number arithmetic, free-variable analysis), your unit tests won't be graded. But we still expect the following:

- You will **indent all unit tests by eight spaces.** This indentation will enable graders to focus on your code.
- You will use unit tests wisely. If a function is simple, do take a minute to validate it with a unit test. If a function is not so simple, develop unit tests in the same way you have done for the past three assignments: one unit test per case in the code.
- If you need debugging help during office hours, we expect that your code will be accompanied by failing unit tests. (If you cannot get your code to typecheck, we will help you do this without unit tests. But if you need help getting code to produce right answers, we will ask to see your unit tests.)

We expect case analysis only when necessary

Case analysis is the enemy. All the more so when it is not necessary. Redundant case analysis is a problem in all levels of programming, but as you are learning ML, it is especially easy to fall into. Redundant case analysis typically manifests in one of two ways:

1) Two cases are present in a `fun`, or `case`, but one is completely subsumed by the other. The most common example is one case to handle the empty list and another case that handles all lists. The empty-list case is often redundant.

   Example:

   ```
   fun append ([], ys) = ys
     | append (xs, ys) = foldr op :: ys xs
   ```

   In this code, the first case is subsumed by the second. It can be eliminated without changing the meaning of the code, and eliminating it typically improves performance.

2) A case analysis is performed where no case analysis is needed.

   ```
   fun sum []        = 0
     | sum (n :: ns) = foldl op + n ns
   ```

   These two cases should be replaced by a single case:

   ```
   fun sum ns = foldl op + 0 ns
   ```

We expect you to examine your code carefully and to remove all redundant case analyses.

We don't expect *written* algebraic laws

We expect you to continue using a systematic design process, but because ML code is so close to algebraic laws, we don't expect you to write algebraic laws separately. If you come to office hours, however, we do expect you to be able to *talk* about algebraic laws or write them on the board.

We expect an acceptable style

Nobody can learn good style in a week. But you can learn to imitate somebody else's style, and we expect you to be judicious about what style you imitate. You have access to books by Ullman, Ramsey, and Harper, and to a technical report by Tofte. These sources are not equally good:

- Ullman provides the most gentle introduction to ML, and he provides the most information about ML. His book is especially good for programmers whose primary experience is in C-like languages. But, to put it politely, Ullman's code is not idiomatic. **Much of what you see from Ullman should not be imitated**.
- Ramsey's code, starting in Chapter 5, is a better guide to what ML should look like. Harper's code is also very good, and Tofte's code is reasonable.

On this assignment, we expect you to devote a *little* effort to good style. Focus on getting your code working first. Then submit it. Then pick up our "Style Guide for Standard ML Programmers", which contains many examples of good and bad style. Edit your code lightly to conform to the style guide, and submit it again.

In the long run, we expect you to master and follow the guidelines in the style guide.

We expect you to remove redundant parentheses

As a novice, you'll be uncertain about where to put parentheses—and you may wind up putting them everywhere. We are fine with parentheses used to disambiguate infix operators, but other redundant parentheses are not OK. To help you find and remove redundant parentheses, we provide a tool called `sml-lint`. We expect you to run

```
sml-lint warmup.sml
sml-lint mlscheme.sml
```

and to remove the parentheses that are named there. (The `sml-lint` program is also run as part of the submission process.)

We expect you to avoid forbidden functions and constructs

While not everybody can learn good style quickly, everybody can learn to avoid the worst faults. In ML, you must avoid these functions and idioms:

- Unlike μScheme, Standard ML provides a `length` function in the initial basis. It is banned. The entire assignment must be solved without using `length`.

  Solutions that use `length` will earn No Credit.

- Use function definition by pattern matching. Do not use the functions `null`, `mynull`, `hd`, and `tl`; use patterns instead.

  Solutions that use `hd` or `tl` will earn No Credit.

- Except for functions given below, *do not define auxiliary functions at top level.* Use `local` or `let`. You will find it useful to use `local` to define functions for use in unit tests.

  Solutions that define auxiliary functions at top level will earn No Credit.

- *Do not use* `open`; if needed, use short abbreviations for common structures. For example, if you want frequent access to the `ListPair` structure, you can write

  ```
  structure LP = ListPair
  ```

  and from there on you can refer to, e.g., `LP.map`.

  Solutions that use `open` may earn No Credit for your entire assignment.

- Unless the problem explicitly says it is OK, do not use any imperative features.

  Unless explicitly exempted, **solutions that use imperative features will earn No Credit.**

Reading comprehension

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

Problems to solve individually

Working on your own, please solve the problems below, which are organized into three sections. Place your solutions in file `warmup.sml`. At the start of each problem, please place a short comment, like

```
(***** Problem A *****)
```

At the very end of your `warmup.sml`, please place the following line:

```
val () = Unit.reportWhenFailures ()  (* put me at the _end_ *)
```

This placement will ensure that if a unit test fails, you are alerted.

To receive credit, your `warmup.sml` file must compile and execute in the Moscow ML system, and your functions must have the right types. At minimum, your code must compile *without warnings or errors*, it must pass `ml-function-check`, it must not have redundant parentheses, and it must not contain any failing unit tests:

```
% /usr/sup/bin/mosmlc -toplevel -I /comp/105/lib warmup.sml
               # DO NOT USE THE -c OPTION
% ml-function-check warmup.sml
% sml-lint warmup.sml
% ./a.out                # runs your Unit tests
```

Revisiting lists (30%)

In this first section, you define functions that are either identical to or very similar to functions you have defined in μScheme.

Defining functions using clauses and patterns

Related Reading for problem A: In *Learning Standard ML* read about Expressions (I, II, and III), Data (I, II, and III), Inexhaustive pattern matches, Types (I), Definitions (III and IV), and Expressions (VIII).

A. Define a function `mynull : 'a list -> bool`, which when applied to a list tells whether the list is empty. Avoid `if`, and make sure the function takes constant time. Do not call any functions from the Standard Basis library. Make sure your function has the same type as the `null` in the Standard Basis.

Lists

Related Reading for problems B to E: In *Learning Standard ML*, in addition to the section noted above, read about Types (III), and Exceptions. You will need to understand lists and pattern matching on lists (see Data III). You may also wish to read the section on Curried Functions.

B. Functions `foldl` and `foldr` are predefined with type

```
('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

They are like the µScheme versions except the ML versions are Curried.

1. Define `reverse : 'a list -> 'a list` using `foldl` or `foldr`.
   (In ML, the reverse function is in the initial basis as `rev`.)

   When you are testing `reverse`, you may get a warning message about "value polymorphism." This message is explained in *Learning Standard ML* (Type pitfall II.)

2. Implement `minlist : int list -> int`, which returns the smallest element of a *nonempty* list of integers. Use `foldl` or `foldr`.

   If given an empty list of integers, your solution must fail (e.g., by `raise Match`).

   Your solution should work regardless of the representation of integers: it should not matter how many bits are used to represent a value of type `int`. (*Hint*: The course solution `max*` from the hofs homework works regardless of the representation of integers. Perhaps you can steal an idea from it.)

   You may find a use for function `Int.min`, which is part of the initial basis of Standard ML.

Do not use recursion in either part of this problem.

C. Define a function `zip: 'a list * 'b list -> ('a * 'b) list` that takes a pair of lists (of equal length) and returns the equivalent list of pairs. If the lengths don't match, raise the exception `Mismatch`, which you must define. Do not call any functions from the Standard Basis Library.

You are welcome to translate a solution from µScheme, but you must either use a clausal definition or write code containing at most one `case` expression. Do not use `if`.

D. Define a function

```
val pairfoldrEq : ('a * 'b * 'c -> 'c) -> 'c -> 'a list * 'b list -> 'c
```

that applies a three-argument function to a pair of lists of equal length, using the same order as `foldr`. Do not call any functions from the Standard Basis Library. If `pairfoldrEq`'s contract is violated (by calling it on lists of unequal lengths), it must raise an exception.

Define a function `ziptoo : 'a list * 'b list -> ('a * 'b) list` which does exactly the same things as `zip` but which uses `pairfoldrEq` for its implementation.

E. Define a function

```
val concat : 'a list list -> 'a list
```

which takes a list of lists of `'a` and produces a single list of `'a` containing all the elements in the correct order. For example,

```
- concat [[1], [2, 3, 4], [], [5, 6]];
> val it = [1, 2, 3, 4, 5, 6] : int list
```

Do not use `if`. You may use functions from the Standard Basis Library, except for `List.concat`—code that uses `List.concat` will earn No Credit.

To get full credit for this problem, your function should use no unnecessary cons cells. Keep in mind the cost of appending two lists.

Note: Function `concat` is closely related to the `flatten` that you implemented in μScheme, but `concat` "goes only one level down."

Revisiting S-expressions (10%)

This section contains just one problem, which asks you to do something new with a concept you have already mastered: use algebraic data types to define a representation of ordinary S-expressions.

Working with a simple algebraic data type

Related Reading for problem F: In *Learning Standard ML*, consult section Data IV (Datatypes).

F. *Ordinary S-expressions*.
The lecture notes define this representation of S-expressions in ML:

```
datatype sx
  = SYMBOL of string
  | NUMBER of int
  | BOOL   of bool
  | SXLIST of sx list
```

Using this representation, define two functions:

- Define function `numbersSx`, which is given a list of numbers and creates an ordinary S-expression:

  ```
  val numbersSx : int list -> sx
  ```

  Use the initial basis effectively.

- Define a function `flattenSyms` that extracts *just* the symbols from an ordinary S-expression:

  ```
  val flattenSyms : sx -> string list
  (* equivalent to uScheme (o ((curry filter) symbol?) flatten) *)
  ```

To write unit tests on functions that return symbolic expressions, try using this string-conversion function:

```
fun sxString (SYMBOL s) = s
  | sxString (NUMBER n) = Int.toString n
  | sxString (BOOL b)   = if b then "true" else "false"
  | sxString (SXLIST sxs) = "(" ^ String.concatWith " " (map sxString sxs) ^ ")"
```

Arithmetic on digit sequences (40%)

In the final individual section, you use constructed data and pattern matching to implement efficient arithmetic on *unbounded* natural numbers. Unlike C and C++, which limit arithmetic to only as many bits as are in a machine word, civilized languages support arithmetic on as many bits as will fit in memory.[2] Every computer scientist should know how this feature is implemented.

In this assignment, we'll implement just addition and subtraction on natural numbers. In later assignments, we'll add multiplication and division, and we'll extend arithmetic to include signed integers. The algorithms are described in *Programming Languages: Build, Prove, and Compare* in Appendix B, which starts on page~S15. (The algorithms are the same ones that you may have learned in elementary school—although now that everyone carries a smartphone, the ability to do arithmetic by hand is definitely a 20th-century skill.)

Arithmetic by pattern matching on constructed data

Our representation of natural numbers is based on the Decimal proof system from the first lesson on program design. That proof system says the following:

- A *digit* is an integer in the range 0 to 9 inclusive.
- Zero is a natural number (rule DecimalZero).
- If m is a natural number and d is a digit, then $10 \times m + d$ is a natural number (rule DecimalNat).

I choose to represent a natural number as a data structure defined by an algebraic data type:

```
datatype nat = ZERO
             | TIMES10PLUS of nat * int
```

(This data structure is equivalent to a list of digits with the least-significant digit first.) The meaning of the data structure is determined by an *abstraction function* and *representation invariants*:

- The abstraction function says that `ZERO` stands for the natural number zero, and if m is a natural number and d is a digit, then `TIMES10PLUS (`m`, `d`)` stands for the natural number $m \times 10 + d$. An abstraction function is conventionally written with a fancy $\mathcal{A}$, but in code comments we use just a plain `A`:

  ```
  A (ZERO)              = 0
  A (TIMES10PLUS (m, d)) = m * 10 + d
  ```

- The representation invariants say that in any value of the form `TIMES10PLUS (`m`, `d`)`,
  - m and d are not both zero.
  - d is a machine integer in the range $0 \leq d < 10$.

The first representation invariant is maintained by the following function, which is called a "smart constructor":

```
fun times10plus (ZERO, 0) = ZERO
  | times10plus (m, d)    = TIMES10PLUS (m, d)
```

[2]Such languages include Haskell, Icon, and Python. But sadly, not Standard ML, because in ML, the so-called "large-integer arithmetic" is optional.

If you like, you can add code to the second case to enforce the invariant on `d`.

Here are a couple of useful special cases:

```
(* times10 : nat -> nat *)
fun times10 n = times10plus (n, 0)

(* natOfDigit : int -> nat *)
fun natOfDigit d = times10plus (ZERO, d)
```

Put the definitions of `nat`*,* `times10plus`*, and the special-case functions into your code.*

You will define functions for conversion, addition, subtraction. Multiplication is extra credit.

For testing, you will find it useful to convert a list of decimal digits to a natural number. You are welcome to use this code:

```
fun flip f (x, y) = f (y, x)

(* natOfDigits : int list -> nat *)
fun natOfDigits ds = foldl (flip times10plus) ZERO ds
```

You may also find it useful to convert a natural number to a string. Here is a function you can pass to `Unit.checkExpectWith`:

```
fun rawNatString ZERO = "ZERO"
  | rawNatString (TIMES10PLUS (m, d)) =
      "(" ^ rawNatString m ^ " * 10 + " ^ Int.toString d ^ ")"
```

Related Reading:

- If needed, review the reading on pattern matching you've already done.
- In *Learning Standard ML*, read the section on datatypes—Data IV. Make sure you understand how to pattern match on constructed values.
- For detailed descriptions of algorithms for addition and subtraction, with examples, read *Programming Languages: Build, Prove, and Compare*, Appendix B, pages S17 to S20.
- To understand how `op` is used in the unit-test examples, consult *Expressions VII: Infix operators as functions* in *Learning Standard ML.*

G. *Natural-number conversions*.
You will convert between natural numbers, machine integers, and strings.

1. Define a function

   ```
   val intOfNat : nat -> int
   ```

   that converts a natural number into a machine integer, or if the natural number is too large, raises `Overflow`. (Use the built-in operators `+` and `*`, which do machine arithmetic and which automatically raise `Overflow` when needed.)

   Example:

   ```
   - intOfNat (natOfDigits [1, 2, 3]);
   > val it = 123 : int
   ```

2. Write a unit test confirming what the example shows: that `intOfNat (natOfDigits [1, 2, 3])` is 123.

3. Define a function

   ```
   val natOfInt : int -> nat
   ```

   that converts a *nonnegative* machine integer into a natural number.

   Example:

   ```
   - natOfInt 2018;
   > val it = TIMES10PLUS(TIMES10PLUS(TIMES10PLUS(TIMES10PLUS(ZERO, 2), 0), 1), 8)
        : nat
   ```

   Use pattern matching, not `if`.

   A nonnegative machine integer is either zero or it has the form $n = 10 \times m + d$. In the second case, d is `(n mod 10)` and m is `(n div 10)`.

4. Write a unit test confirming the `natOfInt` example.

5. Define function `natString`, which converts a `nat` to a string the way we normally write it (with the most significant digit first).

   ```
   val natString : nat -> string
   ```

   Examples:

   ```
   - natString (natOfDigits [3, 2, 1]);
   > val it = "321" : string
   - natString (natOfDigits [2, 0, 1, 8]);
   > val it = "2018" : string
   ```

   Function `natString` must never return an empty string.

   To earn a passing grade, `natString` *must work on 30-digit numbers*. It is safe to use `Int.toString` on a single digit, but if you try to use it on a natural number, the code will fail.

   Hint: Go back to the handout on proof systems for natural numbers. The representation above is based on the Decimal system. But the `natString` function needs to be based on the DecNumeral system. Find a way to deal with the difference.

6. Write a unit test confirming the `natString` example.

Our model solutions take 7 lines of code and 18 lines of (paranoid) unit tests.

H. *Natural-number arithmetic*.
You will add and subtract natural numbers.

1. Define function `carryIntoNat : nat * int -> nat`.

   Function `carryIntoNat` takes a natural number n and a *carry bit* c, and it returns $n + c$. A carry bit is a machine integer that is either 0 or 1.

   Function `carryIntoNat` is defined by these algebraic laws:

```
carryIntoNat (n, 0) == n
carryIntoNat (0, c) == c
carryIntoNat (10 * m + d, 1) ==
     10 * carryIntoNat (m, (d + 1) div 10)
        + ((d + 1) mod 10)
```

To convert these laws into code, you will need to write the natural-number patterns for `0` and for `10 * m + d` as constructed-data patterns `ZERO` and `TIMES10PLUS (m, d)`. *And* you will need to write the natural-number arithmetic on the right-hand side using the smart constructor `times10plus`.

2. Define function `addWithCarry : nat * nat * int -> nat`.

   Function `addWithCarry` takes two natural numbers n_1 and n_2, and a *carry bit* c, and it returns $n_1 + n_2 + c$. To earn a passing grade, *it must be capable of adding 30-digit numbers*, regardless of the number of bits available in a machine integer.

   Function `addWithCarry` is defined by these algebraic laws:

   ```
   addWithCarry (n1, 0, c) = carryIntoNat (n1, c)
   addWithCarry (0, n2, c) = carryIntoNat (n2, c)
   addWithCarry (10 * m1 + d1, 10 * m2 + d2, c) =
      let val d  = (d1 + d2 + c) mod 10
          val c' = (d1 + d2 + c) div 10  (* the ”carry out” *)
      in  10 * addWithCarry (m1, m2, c') + d
      end
   ```

   To convert these laws into code, you will need to write the natural-number patterns as constructed-data patterns, and you will need to write the final operation between `in ... end` using the smart-constructor function.

3. Define function `addNats : nat * nat -> nat`, as follows:

   ```
   fun addNats (n1, n2) = addWithCarry (n1, n2, 0)
   ```

4. Define function `borrowFromNat : nat * int -> nat`.

   Function `borrowFromNat` takes a natural number n and a *borrow bit* b, and it returns $n - b$, provided that $n - b$ is a natural number. If $n - b$ is not a natural number, `borrowFromNat` raises the exception `Negative`, which you will need to define. The borrow bit, like a carry bit, is a machine integer that is either 0 or 1.

   Function `borrowFromNat` is defined by these algebraic laws:

   ```
   borrowFromNat (n, 0) == n
   borrowFromNat (10 * m + 0, 1) == 10 * borrowFromNat (m, 1) + 9
   borrowFromNat (10 * m + d, 1) == 10 * m + (d - 1), where d > 0
   ```

   There is no law for the left-hand side `borrowFromNat (0, 1)`. That's because $0-1$ is not a natural number. Therefore, if your code encounters this case, it should raise the `Negative` exception.

   To convert these laws into code, you will need to write the natural-number patterns as constructed-data patterns, and you will need to write some of the arithmetic on the right-hand side using the smart-constructor function.

5. Define function `subWithBorrow : nat * nat * int -> nat`.

Function `subWithBorrow` takes two natural numbers n_1 and n_2, and a *borrow bit* b, and if $n_1 - n_2 - b$ is a natural number, it returns $n_1 - n_2 - b$. Otherwise it raises the `Negative` exception.

Like `addWithCarry`, `subWithBorrow` must be capable of subtracting 30-digit numbers.

Function `subWithBorrow` is defined by these algebraic laws:

```
subWithBorrow (n1, 0, b) = borrowFromNat (n1, b)
subWithBorrow (10 * m1 + d1, 10 * m2 + d2, b) =
   let val d  = (d1 - d2 - b) mod 10
       val b' = if d1 - d2 - b < 0 then 1 else 0 (* the "borrow out" *)
   in  10 * subWithBorrow (m1, m2, b') + d
   end
```

Alert: These laws assume the Standard ML definition of `mod`, which is not what you get from the hardware. The result of `k mod 10` is always nonnegative.

To convert these laws into code, you will need to write the natural-number patterns as constructed-data patterns, and you will need to write the final operation between `in ... end` using the smart-constructor function.

6. Define function `subNats : nat * nat -> nat`, as follows:

   ```
   fun subNats (n1, n2) = subWithBorrow (n1, n2, 0)
   ```

   Here is a unit test to confirm that subtracting too large a number raises the proper exception: it should raise `Negative` and not `Match`:

   ```
   val () =
     Unit.checkExnSatisfiesWith natString "1 - 5"
     (fn () => subNats (natOfDigits [1], natOfDigits [5]))
     ("Negative", fn Negative => true | _ => false)
   ```

If you trust your conversion functions from the previous problem, you can write unit tests using higher-order functions. Here is an example:

```
fun opsAgree name intop natop n1 n2 =
  Unit.checkExpectWith Int.toString name
  (fn () => intOfNat (natop (natOfInt n1, natOfInt n2)))
  (intop (n1, n2) handle Overflow => 0)
```

Function `opsAgree` has type

```
val opsAgree :
   string -> (int * int -> int) -> (nat * nat -> nat) ->
   int -> int -> unit
```

And it is used as follows

```
val () = opsAgree "123 + 2018" (op +)  addNats 123 2018
val () = opsAgree "2018 - 123" (op -)  subNats 2018 123
val () = opsAgree "2018 * 123" (op * ) mulNats 2018 123
val () = opsAgree "100 - 1   " (op -)  subNats 100 1
```

(Multiplication is for extra credit.)

Our model addition functions total 14 lines of code, not counting unit tests. Our model subtraction functions also total 14 lines of code, not counting unit tests.

Hints:

- Exploit the representation invariant. If a natural number matches the pattern `TIMES10PLUS (_, _)`, the representation invariant guarantees that the number is not zero.
- To maintain the representation invariant, use `TIMES10PLUS` *only* in pattern matching. On the right-hand side of any algebraic law, build natural numbers using the smart constructor `times10plus`.

With a partner: Closures (20%)

As noted above, ML-like languages excel at analyzing and manipulating programs. In the coming month, you will write many functions that analyze expressions; this problem will get you off to a good start. The problem uses the idea of "free variables" which you may remember from the operational-semantics homework, but in the presence of `let` and `lambda` forms, the definition of free variables is much more nuanced than it is in Impcore.

Related Reading for exercise~10: *Build, Prove, and Compare*, section~5.6, which starts on page~322. Focus on the proof system for judgment $y \in \text{fv}(e)$; it is provable exactly when `freeIn e y`, where `freeIn` is the most important function in exercise~10. Also read function `eval` in section~5.3. You will modify the case for evaluating `LAMBDA`.

I (book exercise 10). *Improving closures*.
When a compiler translates a lambda expression, it doesn't store the entire environment in the closure; it stores only the free variables of the lambda expression. Interpreters for languages like Lua and Python work in the same way. You'll implement this code improvement in an interpreter for µScheme.

This problem appears in *Build, Prove, and Compare* as exercise 10 on page~332. You'll solve it in a prelude and four parts:

- The prelude is to get a fresh copy of the book code:

  ```
  git clone homework.cs.tufts.edu:/comp/105/build-prove-compare
  ```

  Now copy the file `build-prove-compare/bare/uscheme-ml/mlscheme.sml` to your working directory. (This file contains all of the interpreter from Chapter 5.) Then make *another* copy and name it `mlscheme-improved.sml`. You will edit `mlscheme-improved.sml`.

- The first part is to implement the free-variable predicate

  ```
      val freeIn : exp -> name -> bool.
  ```

  This predicate tells when a variable appears free in an expression. It implements the proof rules in section 5.6 of the book, which starts on page~322.

  During this part I recommend that you **compile early and often** using

  ```
  /usr/sup/bin/mosmlc -toplevel -I /comp/105/lib mlscheme-improved.sml
  ```

  We also require **unit tests** for `freeIn`. At minimum, write two tests for each short example in the reading-comprehension questions: one for a variable that is free, and one for a variable that appears in the expression but is not free. (If you get these examples right, you have a decent chance of getting the whole problem right.) Additional unit tests for `LET` forms are recommended but not required.

To run the unit tests, you will need to compile and run your interpreter:

```
/usr/sup/bin/mosmlc -toplevel -I /comp/105/lib mlscheme-improved.sml
./a.out -q < /dev/null
```

- The second part is to write a function that takes a pair consisting of a `LAMBDA` body and an environment, and returns a better pair containing the same `LAMBDA` body paired with an environment that contains only the free variables of the `LAMBDA`. (In the book, in exercise 9 starting on page~331, this environment is explained as the *restriction* of the environment to the free variables.) I recommend that you call this function `improve`, and that you give it the type

  ```
  val improve : (name list * exp) * 'a env -> (name list * exp) * 'a env
  ```

- The third part is to use `improve` in the evaluation case for `LAMBDA`, which appears in the book on page 316e. You simply apply `improve` to the pair that is already there, so your improved interpreter looks like this:

  ```
  (* more alternatives for [[ev]] for \uscheme 316e *)
  | ev (LAMBDA (xs, e)) = CLOSURE (improve ((xs, e), rho))
  ```

  If there's a fault in `improve`, your measurements won't mean anything, and your code won't pass our tests. So at this point, you must *test the interpreter on your Scheme homework*.

  The source code you have doesn't include implementations of `record` definitions or short-circuit Booleans, but these forms can be "desugared" into core μScheme. We provide a script for that, so you can test as follows:

  ```
  mosmlc -o improve-test -I /comp/105/lib mlscheme-improved.sml
  desugar-uscheme solution.scm | ./improve-test -q
  ```

- The fourth and final part is to see if it makes a difference. You will compile both versions of the μScheme interpreter using MLton, which is an optimizing, native-code compiler for Standard ML. The compiler requires some annoying bureaucracy, but it compensates by providing native-code speeds.

  The original file, which has no unit tests, can be compiled without bureaucracy:

  ```
  mlton -verbose 1 -output mlscheme mlscheme.sml
  ```

  (If plain `mlton` doesn't work, try `/usr/sup/bin/mlton`.)

  Compiling your improved version requires some bureaucracy to incorporate the `Unit` module. That bureaucracy is incorporated into a script we provide:

  ```
  mlton-with-unit -verbose 1 -output mlscheme-improved mlscheme-improved.sml
  ```

  If you wish to work on your own computer, you'll have to use the old way:

  ```
  mlton -verbose 1 -output mlscheme-improved mlscheme-with-unit.mlb
  ```

  The file `mlscheme-with-unit.mlb` tells MLton to compile your code with our `Unit` module. You will also need files `unit.mlb` and `unit-mlton.sml` from `/comp/105/lib`, and you will have to edit `mlscheme-with-unit.mlb` to refer to your local copy of `unit.mlb`, not the one in `/comp/105/lib`.

Once compiled, you will run both versions and see if the "improvement" is measurable. For measurement, I have provided a script you can use. I also recommend that you compare the performance of the ML code with the performance of the C code in the course directory.

To get a good measurement, you will need to turn off the "CPU throttling" feature that is built into our interpreters. Use the following arcane Unix commands:

```
env BPCOPTIONS=nothrottle time run-exponential-subsets 21 ./mlscheme
env BPCOPTIONS=nothrottle time run-exponential-subsets 21 ./mlscheme-improved
env BPCOPTIONS=nothrottle time run-exponential-subsets 21 /comp/105/bin/uscheme
```

(If you get an error message along the lines of "CPU time exhausted," something is wrong.)

Hints:

- Focus on function `freeIn`. This is the only recursive function and the only function that requires case analysis on expressions. And it is the only function that requires you to understand the concept of free variables. **All** of these concepts are needed for future assignments.

  Once you understand free variables, the coding is relatively easy.

- In Standard ML, the μScheme function `exists?` is called `List.exists`. You'll have lots of opportunities to use it. If you don't use it, you're making extra work for yourself.

  In addition to `List.exists`, you may find uses for `map`, `foldr`, `foldl`, or `List.filter`.

  You might also find a use for these functions, which are already defined for you:

  ```
  fun fst (x, y) = x
  fun snd (x, y) = y

  fun member x =
    List.exists (fn y => y = x)
  ```

- The case for `LETSTAR` is gnarly, and writing it adds little to the experience. Here are two algebraic laws which may help:

  ```
  freeIn (LETX (LETSTAR, [],    e)) y = freeIn e y

  freeIn (LETX (LETSTAR, b::bs, e)) y = freeIn (LETX (LET, [b], LETX (LETSTAR, bs, e))) y
  ```

- It's easier to write `freeIn` if you use nested functions. Use nesting to avoid passing the variable *y*, which rarely changes. You'll see the same technique used in the `eval` and `ev` functions in the chapter, as well as the model solution for `eval-formula` on the `continuations` homework.

- If you can apply what you have learned on the `scheme` and `hofs` assignments, you should be able to write `improve` on one line, without using any explicit recursion.

- Let the compiler help you: **compile early and often**.

- Once you have the interpreter working, **test it** by running it on your solutions from the `scheme`, `hofs`, or `continuation` assignments. (It's possible to get `freeIn` right but to break everything by making a bad mistake in `improve`. Testing on a full solution set will alert you to such issues.)

Our model implementation of `freeIn` is 21 lines of ML.

Extra credit

There are two extra-credit problems: **MULTIPLY** and **VARARGS**. These problems are to be done individually.

MULTIPLY. *Multiplication of natural numbers.*
Define a function

```
val mulNats : nat * nat -> nat
```

that multiplies two natural numbers. Multiplication obeys these algebraic laws:

```
0 * n == 0
n * 0 == 0
(10 * m1 + d1) * (10 * m2 + d2) ==
         d1 * d2 +
         10 * (m1 * d2 + m2 * d1) +
         100 * (m1 * m2)
```

Each of the summands has to be represented as a natural number:

- Number `d1 * d2` can be computed using machine multiplication and `natOfInt`.
- You can multiply `m1 * d2` and `m2 * d1` using `natOfInt` and `mulNats`. The recursive call is guaranteed to terminate because at least one argument is getting smaller.
- You can multiply `m1 * m2` using `mulNats`. The recursive call is guaranteed to terminate because both arguments are getting smaller.

Our model implementation of `mulNats` is seven lines of ML, plus a couple of one-line helper functions.

VARARGS. *Variadic functions in Scheme*.
Extend μScheme to support procedures with a variable number of arguments. This is Exercise 2(a) on page~329 of *Build, Prove, and Compare*. In addition to your implementation, please also include some evidence that the implementation works.

Avoid common mistakes

Don't use `length`, `hd`, or `tl`; they earn **No Credit**.

If you redefine a type that is already in the initial basis, code will fail in baffling ways. (If you find yourself baffled, exit the interpreter and restart it.) If you redefine a function at the top-level loop, this is fine, unless that function captures one of your own functions in its closure.

Example:

```
fun f x = ... stuff that is broken ...
fun g (y, z) = ... stuff that uses 'f' ...
fun f x = ... new, correct version of 'f' ...
```

You now have a situation where **g is broken, and the resulting error is *very* hard to detect**. Stay out of this situation; instead, **load fresh definitions from a file** using the `use` function.

Never put a semicolon after a definition. I don't care if Jeff Ullman does it—don't you do it. It's wrong! Write a semicolon only when you are deliberately using imperative features.

It's a common mistake to become very confused by **not knowing where you need to use `op`**. Ullman covers `op` in Section 5.4.4, page 165.

It's a common mistake to **include redundant parentheses in your code**. To avoid this mistake, use the `sml-lint` tool, and consult the checklist in the section Expressions VIII (Parentheses) in *Learning Standard ML*.

It's a common mistake to do both your pair work and your solo work in the same directory. The submit scripts will balk.

It's not a common mistake, but it can be devastating: when you're writing a type variable, be sure to use an ASCII quote mark, as in `'a`, not with a Unicode right quote mark, as in `’a` (wrong!). Some text editors, web browsers, or Bluetooth keyboards may use or display Unicode without being asked.

It's not a common mistake, but do not copy `Unit.sml` into your submission directory—you won't be able to submit.

What to submit and how to submit it

Submitting your individual work

Please submit a `README` file containing the names of the people with whom you collaborated and a list identifying which problems that you solved—including any extra credit.

For your individual work, please submit file `warmup.sml`. If you have implemented `mulNats`, please include it in `warmup.sml`. If you have done the **VARARGS** extra-credit problem, please submit it as `varargs.sml`.

In comments at the top of your `warmup.sml` file, please include your name and the names of any collaborators, and a note about any extra-credit work you have done.

As soon as you have a `warmup.sml` file, run `submit105-ml-solo` to submit a preliminary version of your work. As you edit your files, keep submitting; we grade only the last submission.

Submitting your improved μScheme interpreter

For your your improved μScheme interpreter, which you may have done with a partner, please submit the file `mlscheme-improved.sml`, using the script `submit105-ml-pair`. If you worked with a partner, **just one of you should submit.** The submission counts for both partners.

How your work will be evaluated

The criteria are mostly the same as for the `scheme` and `hofs` assignments, but because the language is different, we'll be looking for indentation and layout as described in the Style Guide for Standard ML Programmers.