Core ML

COMP 105 Assignment

Due Tuesday, March 6, 2018 at 11:59PM

Contents

Overview 2

Prelude 2
  Setup 2
  The initial basis 2
  Things you need to review before starting 3
  How to develop an acceptable style 3
  Type checking 4
  Unit testing 4
  What we expect from unit tests 4
  Redundant case analysis 5
  Dire warnings 5

Reading comprehension (10%) 6

Programming problems to solve individually (75%) 8
  How to organize your code 8
  The problems 8
    Defining functions using clauses and patterns 8
    Lists 9
    Arithmetic by pattern matching on lists 10
  Exceptions 13
  Algebraic data types 15
  An immutable, persistent alternative to linked lists 16

One problem you can do with a partner (15%) 17

Extra credit 20
  MULTIPLY 20
  FIVES 20
  VARARGS 21

Make sure your solutions have the right types 21

Avoid other common mistakes 22
What to submit and how to submit it  

- Submitting your individual work ............................ 23  
- Submitting your improved μScheme interpreter .................. 24

How your work will be evaluated 24

NEXT TIME: YOU ABSOLUTELY MUST RUN THE ML SANITY CHECKER (WHICH NEEDS TO FUNCTORIZE ENVBOTH, BY THE WAY.)

Overview

The purpose of this assignment is to help you get acclimated to programming in Standard ML, which you will use in the next few weeks to implement type systems and lambda calculus. The assignment has three parts:

• To begin, you will answer some questions about reading.
• On your own, you will write many small exercises.
• 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 this assignment, you will be ready to tackle serious programming tasks in Standard ML.

Prelude

Setup

COMP 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 2.10-3 (Tufts University, April 2012)
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 comp105-grades@cs.tufts.edu.

For the large problem, we recommend a native-code compiler called mlton (pronounced “Milton”).

The 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 quite 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\(^{3}\), bool, int, string, and order\(^{4}\)
- Modules List\(^{5}\) and Option\(^{6}\), 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 ML Learning Guide\(^7\).

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

```plaintext
- 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\(^8\).

**Things you need to review before starting**

We provide a guide to *Learning Standard ML*\(^9\). And pages 664–665 in *Build, Prove, and Compare* contain a large table that may help you translate μScheme code into core ML. Skim these materials before starting, so you know what is there. *Learning Standard ML* will guide you to other reading.

**How to develop an acceptable style**

*Learning Standard ML* refers to you books by Ullman, Ramsey, and Harper, and to a technical report by Tofte. 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.

Focus on getting your code working first. Then submit it. Then pick up our “Style Guide for Standard ML Programmers”\(^10\), 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\(^11\).

---

\(^{3}\)http://sml-family.org/Basis/option.html#SIG:OPTION.option:TY
\(^{4}\)http://sml-family.org/Basis/general.html#SIG:GENERAL.order:TY
\(^{5}\)http://sml-family.org/Basis/list.html#List:STR:SPEC
\(^{6}\)http://sml-family.org/Basis/option.html#Option:STR:SPEC
\(^{7}\)../readings/ml.html#unit-testing
\(^{8}\)http://sml-family.org/Basis/text-io.html#SIG:TEXT_IO.inputLine:VAL
\(^{9}\)../readings/ml.pdf
\(^{10}\)../handouts/mlstyle.pdf
\(^{11}\)../handouts/mlstyle.pdf
Type checking

Standard ML is a language designed for production, not teaching. It does not come with built-in check-expect or other unit-testing support. It does, however, support some checking of types. In particular, you can check the type of an identifier by rebinding the identifier using an explicit type. There are examples below, and you can run them against your code by running

% ml-sanity-check warmup.sml

Unit testing

Although Standard ML does not have check-expect and friends built in, we can do a lot with higher-order functions. Here are some examples of tests written with our Unit module:

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 fire up mosml, load the Unit module, and paste in this code (followed by a semicolon), you’ll see these reports:

$ mosml -P full -I /comp/105/lib
Moscow ML version 2.10-3 (Tufts University, April 2012)
Enter ‘quit();’ to quit.
- load “Unit”;
> val it = () : unit
- val () = ... ;
In test ‘2 is false’, expected value false but got true
One of two internal Unit tests passed.

You’ll use Unit.checkExpectWith to write your own unit tests. You’ll also use Unit.checkAssert and Unit.checkExnWith. The details are in the ML Learning Guide\textsuperscript{12}.

What we expect from 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:

\textsuperscript{12}../readings/ml.html#unit-testing
• You will **indent all unit tests by eight spaces.** This indentation will enable graders to focus on your code. (I wish I had thought of this earlier.)

• 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, you will have 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 demand to see your unit tests.)

**Redundant case analysis**

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.

   NEEDS AN EXAMPLE HERE.

**Dire warnings**

There some functions and idioms that you must avoid. Code violating any of these guidelines will earn **No Credit.**

• When programming with lists, it is rarely necessary or desirable to use the `length` function. The entire assignment can and should 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`, `hd`, and `tl`; use patterns instead.

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

• Do not define auxiliary functions at top level. Use `local` or `let`.

   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 (10%)**

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. Read section 5.1 of Harper about tuple types and tuple patterns. Also look at the list examples in sections 9.1 and 9.2 of Harper.

Now consider the pattern \((x::y::zs, w)\). For each of the following expressions, tell whether the pattern matches the value denoted. If the pattern matches, say what values are bound to the four variables \(x\), \(y\), \(zs\), and \(w\). If it does not match, explain why not.

(a) \(([1, 2, 3], ("COMP", 105))\)
(b) \((("COMP", 105), [1, 2, 3])\)
(c) \((("COMP", 105)), (1, 2, 3))\)
(d) \((("COMP", "105"), true)\)
(e) \([true, false], 2.718281828)\)

Answers here:

(a)
(b)
(c)
(d)
(e)

You are now starting to be ready to use pattern matching.

2. Look at the clausal function definition of outranks on page 83 of Harper. Using the clausal definition enables us to avoid nested case expressions such as we might find in Standard ML or μML, and it enables us to avoid nested if expressions such as we might find in μScheme. This particular example also collapses multiple cases by using the “wildcard pattern” _._

A wildcard by itself can match anything, but a wildcard in a clausal definition can match only things that are not matched by preceding clauses. Answer these questions about the wildcards in outranks:

---

13 `cqs.ml.txt`
(a) In the second clause, what three suits can the _ match?

→

(b) In the fifth clause, what suits can the _ match?

→

(c) In the eighth and final clause, what suits can the _ match?

→

You are now ready to match patterns that combine tuples with algebraic data types.

3. In Ramsey’s chapter 5, the eval code for applying a function appears in code chunk 365d. In evaluating \( \text{APPLY}(f, \text{args}) \), if expression \( f \) does not evaluate to either a primitive function or a closure, the code raises the \textit{RuntimeError} exception.

(a) Show a piece of μScheme code that would, when evaluated, cause chunk 365d to raise the \textit{RuntimeError} exception.

→

(b) When exception \textit{RuntimeError} is raised, what happens from the user’s point of view?

You are now ready to write \textit{zip} and to write environment functions that use exceptions.

4. “Free” variables are those that are not bound to a value in the current scope. You can find a longer discussion and precise definition in section 5.11 of \textit{Build, Prove, and Compare}, which starts on page 376. Read the section and identify the free variables of the following expressions:

(a) Free variables of \( \lambda(x) \lambda(y) (\text{equal? } x y) \)

→

(b) Free variables of \( \lambda(y) (\text{equal? } x y) \)

→

(c) Free variables of

\[
\lambda(s1 \ s2)
\begin{align*}
& (\text{if} (\text{or} (\text{atom? } s1) (\text{atom? } s2)) \\
& \quad (= s1 s2) \\
& \quad (\text{and} (\text{equal? } (\text{car } s1) (\text{car } s2)) \\
& \quad \quad (\text{equal? } (\text{cdr } s1) (\text{cdr } s2)))) \\
& \end{align*}
\]

→

You are now ready to improve the μScheme interpreter, which you can do with a partner. You and your partner will turn your answers to parts (a) and (b) into unit tests.
Programming problems to solve individually (75%)

How to organize your code

Most of your solutions will go into a single file: `warmup.sml`. But to practice using exceptions, you’ll implement environments in two different ways, and you’ll write code that works with both implementations. Those solutions will go in files `envdata.sml`, `envfun.sml`, and `envboth.sml`. You’ll also want to download `envdata-test.sml` and `envfun-test.sml`.

At the start of each problem, please label it with a short comment, like

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

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

```ml
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. For example, we must be able to compile your code without warnings or errors. The following three commands should test all of your code:

```bash
% /usr/sup/bin/mosmlc -toplevel -I /comp/105/lib -c warmup.sml
% test-env-data
% test-env-fun
```

The problems

Working on your own, please solve the following problems:

Defining functions using clauses and patterns

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

**A.** Write the function `null`, which when applied to a list tells whether the list is empty. Avoid `if`, and make sure the function takes constant time. Make sure your function has the same type as the `null` in the Standard Basis.

**B.** Define a function `firstVowel` that takes a list of lower-case letters and returns `true` if the first character is a vowel (aeiou) and `false` if the first character is not a vowel or if the list is empty. Use the wildcard symbol `_` whenever possible, and avoid `if`.

---

16. /envdata-test.sml
17. /envfun-test.sml
18. ../readings/ml.html
Lists

Related Reading for problems C to F: In Learning Standard ML, apart from the section noted above, read about Types (III), and Exceptions. For this section, you will need to understand lists and pattern matching on lists well (see Data III). You may also wish to read the section on Curried Functions.

C. Functions foldl and foldr are predefined with type

\((a \times b \rightarrow b) \rightarrow b \rightarrow a\ list \rightarrow b\)

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

1. Implement rev (the function known in μScheme as reverse) using foldl or foldr.

2. Implement minlist, which returns the smallest element of a non-empty list of integers. Use foldl or foldr.

   If given an empty list of integers, your solution can 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.)

   Do not use recursion in either part of this problem.

D. Define a function \(zip\): \(a\ list \times b\ list \rightarrow (a \times b)\ list\) 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 will have to define.

   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.

E. Define a function

\[\text{val pairfoldr : (a \times b \times c \rightarrow c) \rightarrow c \rightarrow a\ list \times b\ list \rightarrow c}\]

that applies a three-argument function to a pair of lists of equal length, using the same order as foldr.

Define a function ziptoo : \(a\ list \times b\ list \rightarrow (a \times b)\ list\) which does exactly the same things as zip but which uses pairfoldr for its implementation.

F. Define a function

\[\text{val concat : a\ list\ list \rightarrow a\ list}\]

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

\[- \text{concat [[1], [2, 3, 4], [], [5, 6]];}
\]

\[- > \text{val it} = [1, 2, 3, 4, 5, 6] : \text{int}\ list\]
Do not use \texttt{if}. Do not simply call \texttt{List.concat}—you will earn No Credit.

To get full credit for this problem, your function should use no unnecessary cons cells.

\textbf{Note:} Function \texttt{concat} is related to the \texttt{flatten} that you implemented in μScheme, but \texttt{concat} “goes only one level down.”

\textbf{Arithmetic by pattern matching on lists}

Languages like C and C++ enable you to do arithmetic only on as many bits as are in a machine word. More civilized languages allow arithmetic on as many bits as will fit in all of memory.\textsuperscript{21} Every computer scientist should know how this feature is implemented. In this assignment, we’ll implement arithmetic on natural numbers only, where a natural number is represented as a list of digits.

Our representation is based on the Decimal proof system from the proof-systems handout\textsuperscript{22}. To make the rules readable on the web and not just in the printout, I recapitulate them in informal English:

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

I choose to represent a natural number as a list of digits, with the \textit{least-significant digit first}. This representation is especially good for arithmetic:

- The natural number zero is represented by the empty list.
- You will match zero with the pattern \([\ ]\).
- The natural number \(10 \times m + d\) is represented by the list \((d :: ds)\), where \textit{ds} is the representation of the natural number \textit{m}.
- You will match number \(10 \times m + d\) with the pattern \((d :: m)\).

To memorialize our commitment to this representation, put this \textit{type abbreviation} in your source code:

\begin{verbatim}
type nat = int list
\end{verbatim}

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

\textbf{Related Reading:} The reading on pattern matching you’ve already done. A short note on arithmetic\textsuperscript{23}, which is excerpted from a book chapter in progress. Finally, to understand how \texttt{op} is used in the unit-test examples, consult \textit{Expressions VII: Infix operators as functions}\textsuperscript{24} in Learning Standard ML.

\textbf{G. Natural-number conversions.}

You will convert between natural numbers, machine integers, and strings.

1. Define a function

\begin{verbatim}
val intOfNat : nat -> int
\end{verbatim}

\textsuperscript{21}By this criterion, Standard ML is not civilized, because the so-called “large-integer arithmetic” is optional.

\textsuperscript{22}../handouts/natproofs.pdf

\textsuperscript{23}../handouts/short-arith.pdf

\textsuperscript{24}../readings/ml.html#expressions-vii-infix-operators-as-functions
that converts a list of digits into a machine integer, or if the number represented by the list of digits is too large, raises overflow. (You will use the built-in operators + and *, which do machine arithmetic and which automatically raise overflow when needed.)

Example:
- intOfNat [3, 2, 1];
> val it = 123 : int (* Note: Digits are reversed! *)

2. Write a unit test confirming what the example shows: that intOfNat [3, 2, 1] 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 = [8, 1, 0, 2] : int list

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 [3, 2, 1];
  > val it = "123" : string
- natString [8, 1, 0, 2];
  > val it = "2018" : string

For full credit, write natString without recursion. Instead, use Int.toString, String.concat, rev, map, and o.

6. Write a unit test confirming the natString example.

My solutions take 5 lines of code and 12 lines of unit tests.

H. Natural-number arithmetic.

You will add and subtract natural numbers.

1. Define function carryIntoNat : nat * int -> nat. This function takes a natural number \( n \) and a carry bit \( c \), and it returns \( n + c \). A carry bit is either 0 or 1.

The function is defined by these algebraic laws:

\[
\begin{align*}
\text{carryIntoNat} (n, 0) &= n \\
\text{carryIntoNat} (0, c) &= c \\
\text{carryIntoNat} (10 \times m + d, 1) &= \\
\end{align*}
\]
10 * carryIntoNat (m, (d + 1) div 10) + ((d + 1) mod 10)

To convert these laws into code, you will need to write the patterns for 0 and for 10 * m + d into list patterns [] and (d :: m).

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

The function is defined by these algebraic laws:

\[
\begin{align*}
\text{addWithCarry} (n_1, 0, c) &= \text{carryIntoNat} (n_1, c) \\
\text{addWithCarry} (0, n_2, c) &= \text{carryIntoNat} (n_2, c) \\
\text{addWithCarry} (10 * m_1 + d_1, 10 * m_2 + d_2, c) &= \\
&\quad \text{let } d = (d_1 + d_2 + c) \mod 10 \\
&\quad \text{val } c' = (d_1 + d_2 + c) \div 10 \text{ (* the "carry out" *)} \\
&\quad \text{in } 10 * \text{addWithCarry} (m_1, m_2, c') + d \\
\end{align*}
\]

To convert these laws into code, you will need to write the natural-number patterns as list patterns.

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

\[
\begin{align*}
\text{fun addNats} (n_1, n_2) &= \text{addWithCarry} (n_1, n_2, 0) \\
\end{align*}
\]

4. Define function borrowFromNat : nat * int -> nat. This function 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, \( \text{borrowFromNat} \) raises the exception \text{Negative}, which you will need to define.

The function is defined by these algebraic laws:

\[
\begin{align*}
\text{borrowFromNat} (n, 0) &= n \\
\text{borrowFromNat} (10 * m + 0, 1) &= 10 * \text{borrowFromNat} (m, 1) + 9 \\
\text{borrowFromNat} (10 * m + d, 1) &= 10 * m + (d - 1), \text{ where } d > 0 \\
\end{align*}
\]

Notice there is no law for the left-hand side \( \text{borrowFromNat} (0, 1) \). That's because \( 0 - 1 \) is not a natural number—so if your code encounters this case, it should raise the \text{Negative} exception.

To convert these laws into code, you will need to write the natural-number patterns as list patterns.

5. Define function subWithBorrow : nat * nat * int -> nat. This function 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 \text{Negative} exception.

Like \( \text{addWithCarry} \), \( \text{subWithBorrow} \) must be capable of subtracting 30-digit numbers.

The function is defined by these algebraic laws:

\[
\begin{align*}
\text{subWithBorrow} (n_1, 0, b) &= \text{borrowFromNat} (n_1, b) \\
\text{subWithBorrow} (10 * m_1 + d_1, 10 * m_2 + d_2, b) &= \\
&\quad \text{let } d = (d_1 + d_2 - b) \mod 10 \\
&\quad \text{val } b' = \text{if } d_1 + d_2 - b < 0 \text{ then } 1 \text{ else } 0 \text{ (* the "borrow out" *)} \\
&\quad \text{in } 10 * \text{subWithBorrow} (m_1, m_2, b') + d \\
\end{align*}
\]
Alert: These laws assume the Standard ML definition of \texttt{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 list patterns.

6. Define function \texttt{subNats : nat * nat -> nat}, as follows:

\[
\text{fun subNats (n1, n2) = subWithBorrow (n1, n2, 0)}
\]

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

\[
\text{val () = Unit.checkExnSatisfiesWith natString "1 - 5"}
\]
\[
\text{(fn () => subNats ([1], [5]))}
\]
\[
\text{("Negative", fn Negative => true | _ => false)}
\]

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

\[
\text{fun opsAgree name intop natop n1 n2 =}
\]
\[
\text{Unit.checkExpectWith Int.toString name}
\]
\[
\text{(fn () => intOfNat (natop (natOfInt n1, natOfInt n2)))}
\]
\[
\text{(intop (n1, n2))}
\]

This function has type

\[
\text{val opsAgree : string -> (int * int -> int) -> (nat * nat -> nat) ->}
\]
\[
\text{int -> int -> unit}
\]

And it is used as follows

\[
\text{val () = opsAgree "123 + 2018" (op +) addNats 123 2018}
\]
\[
\text{val () = opsAgree "2018 - 123" (op -) subNats 2018 123}
\]
\[
\text{val () = opsAgree "2018 * 123" (op *) mulNats 2018 123}
\]
\[
\text{val () = opsAgree "100 - 1 " (op -) subNats 100 1}
\]

(Multiplication is for extra credit.)

My addition functions total 13 lines of code, not counting unit tests. My subtraction functions also total 13 lines of code, not counting unit tests.

Exceptions

Related Reading for problems I: In \textit{Learning Standard ML}\textsuperscript{25}, read the section on Curried functions. Read the sections on Types (III) and Data (IV). Make sure you understand the difference between types and datatypes. Read the section on Exceptions, and make sure you know both how to raise and how to handle an exception.

1. Environments with exceptions.

\textsuperscript{25} ../readings/ml.html
1. In file envdata.sml, write definitions of a type `a env and functions

```ml
type 'a env = (* you fill in this part *)
exception NotFound of string
val emptyEnv : 'a env = (* ... *)
val bindVar : string * 'a * 'a env -> 'a env = (* ... *)
val lookup : string * 'a env -> 'a = (* ... *)
```

such that you can use `a env for a type environment or a value environment. On an attempt to look
up an identifier that doesn’t exist, raise the exception NotFound. Don’t worry about efficiency.

You can test your work interactively in mosml:

```
Moscow ML version 2.10-3 (Tufts University, April 2012)
Enter ‘quit();’ to quit.
- load "Unit";
- use "envdata.sml";
- use "envdata-test.sml";
```

If all is well, you’ll get a bunch of messages about names and types, but no errors.

2. In file envfun.sml, do the same, except make type `a env = string -> 'a, and let

```ml
fun lookup (name, rho) = rho name
```

As above, you can test using mosml and envfun-test.sml.

3. In file envboth.sml, define a function

```ml
val isBound : string * 'a env -> bool
```

that works with both representations of environments. That is, write a single function that works
regardless of whether environments are implemented as lists or as functions. You will need im-
perative features, like sequencing (the semicolon). Don’t use if.

Test your code by running the following shell commands:

```
cat envdata.sml envboth.sml | mosml -P full -I /comp/105/lib
```

4. Also in file envboth.sml, define a function

```ml
val extendEnv : string list * 'a list * 'a env -> 'a env
```

that takes a list of variables and a list of values and adds the corresponding bindings to an envi-
ronment. It should work with both representations. Do not use recursion. Hint: you can do it in
two lines using the higher-order list functions defined above. You will have to copy the relevant
functions into envboth.sml.

Test your code by running the following shell commands:

```
cat envdata.sml envboth.sml | mosml -P full -I /comp/105/lib
```

```
Algebraic data types

(For this problem and all the remaining problems, we are back to putting the solutions into warmup.sml.)

Related Reading for problem J: In Learning Standard ML, read the section on datatypes—Data IV. Make sure you understand how to pattern match on constructed values.

J. Search trees.
ML can easily represent binary trees containing arbitrary values in the nodes:

datatype 'a tree = NODE of 'a tree * 'a * 'a tree |
| LEAF

To make a search tree, we need to compare values at nodes. The standard idiom for comparison is to define a function that returns a value of type order. As discussed in Ullman, page 325, order is predefined by

datatype order = LESS | EQUAL | GREATER (* do not include me in your code *)

Because order is predefined, if you include it in your program, you will hide the predefined version (which is in the initial basis) and other things may break mysteriously. So don’t include it.

We can use the order type to define a higher-order insertion function by, e.g.,

fun insert cmp =
  let fun ins (x, LEAF) = NODE (LEAF, x, LEAF)
    | ins (x, NODE (left, y, right)) =
      (case cmp (x, y)
       of LESS => NODE (ins (x, left), y, right)
       | GREATER => NODE (left, y, ins (x, right))
       | EQUAL => NODE (left, x, right))
  in ins end

This higher-order insertion function accepts a comparison function as argument, then returns an insertion function. (The parentheses around case aren’t actually necessary here, but I’ve included them because if you leave them out when they are needed, you will be very confused by the resulting error messages.)

We can use this idea to implement polymorphic sets in which we store the comparison function in the set itself. For example,

datatype 'a set = SET of ('a * 'a -> order) * 'a tree
fun nullset cmp = SET (cmp, LEAF)

- Define a function
  val addelt : 'a * 'a set -> 'a set
  that adds an element to a set.

- Define a function
  val treeFoldr : ('a * 'b -> 'b) -> 'b -> 'a tree -> 'b

\[/text]
that folds a function over every element of a tree, rightmost element first. Calling `treeFoldr (op ::) [] t` should return the elements of `t` in order. Write a similar function:

```plaintext
val setFold : ('a * 'b -> 'b) -> 'b -> 'a set -> 'b
```

The function `setFold` should visit every element of the set exactly once, in an unspecified order.

**An immutable, persistent alternative to linked lists**

**Related Reading** for problem K: In *Learning Standard ML*⁷, read the section on datatypes—Data IV. Make sure you understand how to pattern match on constructed values.

**K.** For this problem I am asking you to define your own representation of a new abstraction: the list with finger. A list with finger is a nonempty sequence of values, together with a “finger” that points at one position in the sequence. The abstraction provides constant-time insertion and deletion at the finger.

**This is a challenge problem.** The other problems on the homework all involve old wine in new bottles. To solve this problem, you have to think of something new.

1. Define a representation for type `'a flist`. (Before you can define a representation, you will want to study the rest of the parts of this problem, plus the test cases.) Document your representation by saying, in a short comment, what sequence is meant by any value of type `'a flist`.

2. Define function

```plaintext
val singletonOf : 'a -> 'a flist
```

which returns a sequence containing a single value, whose finger points at that value.

3. Define function

```plaintext
val atFinger : 'a flist -> 'a
```

which returns the value that the finger points at.

4. Define functions

```plaintext
val fingerLeft : 'a flist -> 'a flist
val fingerRight : 'a flist -> 'a flist
```

Calling `fingerLeft xs` creates a new list that is like `xs`, except the finger is moved one position to the left. If the finger belonging to `xs` already points to the leftmost position, then `fingerLeft xs` should raise the same exception that the Basis Library raises for array access out of bounds. Function `fingerRight` is similar. Both functions must run in constant time and space. Please think of these functions as “moving the finger”, but remember no mutation is involved. Instead of changing an existing list, each function creates a new list.

5. Define functions

```plaintext
val deleteLeft : 'a flist -> 'a flist
val deleteRight : 'a flist -> 'a flist
```

⁷ ../readings/ml.html
Calling `deleteLeft xs` creates a new list that is like `xs`, except the value `x` to the left of the finger has been removed. If the finger points to the leftmost position, then `deleteLeft` should raise the same exception that the Basis Library raises for array access out of bounds. Function `deleteRight` is similar. Both functions must run in constant time and space. As before, no mutation is involved.

6. Define functions
   
   ```
   val insertLeft : 'a * 'a flist -> 'a flist
   val insertRight : 'a * 'a flist -> 'a flist
   ```
   
   Calling `insertLeft (x, xs)` creates a new list that is like `xs`, except the value `x` is inserted to the left of the finger. Function `insertRight` is similar. Both functions must run in constant time and space. As before, no mutation is involved. (These functions are related to “cons”.)

7. Define functions
   
   ```
   val ffoldl : ('a * 'b -> 'b) -> 'b -> 'a flist -> 'b
   val ffoldr : ('a * 'b -> 'b) -> 'b -> 'a flist -> 'b
   ```
   
   which do the same thing as `foldl` and `foldr`, but ignore the position of the finger.

Here is a simple test case, which should produce a list containing the numbers 1 through 5 in order. You can use `ffoldr` to confirm.

   ```
   val test = singletonOf 3
   val test = insertLeft (1, test)
   val test = insertLeft (2, test)
   val test = insertRight (4, test)
   val test = fingerRight test
   val test = insertRight (5, test)
   ```

You'll want to test the `delete` functions as well.

*Hints:* The key is to come up with a good representation for “list with finger.” Once you have a good representation, the code is easy: over half the functions can be implemented in one line each, and no function requires more than two lines of code.

**One problem you can do with a partner (15%)**

The goal of this problem is to give you practice working with an algebraic data type that plays a central role in programming languages: expressions. In the coming month, you will write many functions that consume expressions; this problem will get you off to a good start. It will also give you a feel for the kinds of things compiler writers do.

The problem is numbered 2 because that’s the problem number in the book. You won’t be doing exercise 1, so you’re not missing anything

*Related Reading* for exercise 2: Ramsey, section 5.11, which starts on page 376. Focus on the proof system for judgment $y \in \text{fv}(e)$; it is provable exactly when $\text{freeIn} e y$, where `freeIn` is the most important function in exercise 2. Also read function `eval` in section 5.4. You will modify the case for evaluating `LAMBDA`. 
2. Improving closures.

When a compiler translates a lambda expression, a compiler doesn’t need to store an entire environment in a closure; it only needs to store the free variables of the lambda expression. This problem appears in Ramsey’s book as exercise 2 on page 383, and you’ll solve it in a prelude and four parts:

- The prelude is to go to your copy of the book code and copy the file `bare/uscheme-ml/mlscheme.sml` to your working directory. (This code 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

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

  This predicate tells when a variable appears free in an expression. It implements the proof rules in section 5.11 of the book, which starts on page 376.

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

  ```bash
  /usr/sup/bin/mosmlc -c -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. Unit tests for `LET` forms are recommended but not required.

- 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 1 starting on page 382, 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

  ```ml
  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 365c. You simply apply `improve` to the pair that is already there, so your improved interpreter looks like this:

  ```ml
  (* more alternatives for [[ev]] for \
scheme 365c *)
  | ev (LAMBDA (xs, e)) = ( errorIfDups ("formal parameter", xs, "lambda")
  | ; CLOSURE (improve ((xs, e), rho))
  ```

- 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:

  ```bash
  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.

  ```bash
  copy-105-ml-files-here
  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. If you wish to do this on your own computer, 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.

In order to run the measurement, you will need to patch your source code to turn of the “CPU throttling” feature. To do this, run these patch commands:

```
patch mlscheme.sml /comp/105/build-prove-compare/throttle-patch.txt
patch mlscheme-improved.sml /comp/105/build-prove-compare/throttle-patch.txt
```

A successful patch produces output something like the following:

```
patching file mlscheme-improved.sml
Hunk #1 succeeded at 976 (offset 1 line).
Hunk #2 succeeded at 994 (offset 1 line).
```

Once you have patched both interpreters, you should be able to recompile them and then run them without triggering a “CPU time exhausted” error. You’ll need the following arcane Unix command:

- env BPCOPTIONS=nothrottle time run-exponential-arg-max 22 ./mlscheme
- env BPCOPTIONS=nothrottle time run-exponential-arg-max 22 ./mlscheme-improved
- env BPCOPTIONS=nothrottle time run-exponential-arg-max 22 /comp/105/bin/uscheme

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. Understanding free variables is hard, but once you understand, the coding is 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 (y, []) = false
| member (y, z::zs) = y = z orelse member (y, zs)
```

- 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. Mostly the variable `y` doesn’t change, so you needn’t pass it everywhere. You’ll see the same technique used in the `eval` and `ev` functions in the chapter, as well as the model solution for `eval` 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.**

My implementation of `freeIn` is 21 lines of ML.

**Extra credit**

There are three extra-credit problems: **MULTIPLY, FIVES, and VARARGS.**

**MULTIPLY**

Define a function

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

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

\[
\begin{align*}
0 \times n & = 0 \\
n \times 0 & = 0 \\
(10 \times m_1 + d_1) \times (10 \times m_2 + d_2) & = \\
& d_1 \times d_2 + \\
& 10 \times (m_1 \times d_2 + m_2 \times d_1) \\
& 100 \times (m_1 \times m_2)
\end{align*}
\]

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

• Number \(d_1 \times d_2\) can be computed using `natOfInt`.
• You can multiply \(m_1 \times d_2\) and \(m_2 \times d_1\) using `natOfInt` and `mulNats`.
• You can multiply \(m_1 \times m_2\) using `mulNats`.
• If number \(p\) is represented as a list of digits \(ds\), then \(10 \times p\) is represented as \((0 :: ds)\).

My implementation of `mulNats` is six lines of ML.

**FIVES**

Consider the class of well-formed arithmetic computations using the numeral 5. These are expressions formed by taking the integer literal 5, the four arithmetic operators +, -, *, and /, and properly placed parentheses. Such expressions correspond to binary trees in which the internal nodes are operators and every leaf is a 5. If you enumerate all such expressions, you can answer these questions:

• What is the smallest positive integer that cannot be computed by an expression involving exactly five 5’s?
• What is the largest prime number that can be computed by an expression involving exactly five 5’s?
• Exhibit an expression that evaluates to that prime number.

Write an ML function `reachable` of type

```plaintext
('a * 'a -> order) * ('a * 'a -> 'a) list -> 'a -> int -> 'a set
```

such that `reachable (Int.compare, [op +, op -, op *, op div]) 5 5` computes the set of all integers computable using the given operators and exactly five 5's. (You don’t have to bother giving the answers to the questions above; the first two are easily gotten with `setFold`.) My solution is under 20 lines of code. Such brevity is possible only because I rely heavily on the `setFold`, `nullset`, `addelt`, and `pairfoldr` functions defined earlier.

**Hints:**

• In order to be able to use `Int.compare` interactively, you will either have to run `mosml -P full` or else tell Moscow ML interactively to load "Int";

• Begin your function definition this way:

```plaintext
fun reachable (cmp, operators) five n =
 (* produce set of expressions reachable with exactly n fives *)
```

• Use dynamic programming\(^{28}\).

• Create a list of length \(k-1\) in which element \(i\) is a set containing all the integers that can be computed using exactly \(i\) elements. Now compute the \(k\)th element of the list by combining 1 with \(k-1\), 2 with \(k-2\), etcetera.

• Try doing the above by passing a list and its reverse, then use `pairfoldr` with a suitable function.

• The initial list contains a set with exactly one element (in the example above, 5).

• Make sure your solution has the completely general type given above, so you could use it with different operations and with different representations of numbers.

**VARARGS**

Extend μScheme to support procedures with a variable number of arguments. This is Exercise 8 on page 385.

**Make sure your solutions have the right types**

On this assignment, it is a very common mistake to define functions of the wrong type. You can protect yourself a little bit by running the script `ml-sanity-check`, which we provide. It loads the following declarations after loading your solution:

```plaintext
(* first declaration for sanity check *)
val firstVowel : char list -> bool = firstVowel
val null : 'a list -> bool = null
val rev : 'a list -> 'a list = rev
```

\(^{28}\)../readings/dynamic.html
Avoid other common mistakes

It’s a common mistake to use any of the functions length, hd, and tl. Instant 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

I don’t promise to have all the functions and their types here—for example, this list includes only functions from warmup.sml, not functions in envdata.sml, envfun.sml, or envboth.sml. The ml-sanity-check script will help you, but making sure that every function has the right type is your job, not mine.
is fine, unless that function captures one of your own functions in its closure.

Example:

```ml
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, but don’t you do it—it’s wrong! You should have a semicolon only if 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 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. Some text editors or web browsers 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**

There is no README file for this assignment.

**Submitting your individual work**

For your individual work, please submit the files cqs.ml.txt, warmup.sml, envdata.sml, envfun.sml, and envboth.sml. If you have implemented mulNats, please include it in warmup.sml. If you have done either of other the extra-credit problems, please submit them as varargs.sml or fives.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, create empty files envdata.sml, envfun.sml, and envboth.sml, and 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.

---

29 ../readings/ml.html#expressions-viii-parentheses
Submitting your improved μScheme interpreter

For 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.

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 Programmers30.

30../handouts/mlstyle.pdf