Core ML

COMP 105 Assignment

Due Wednesday, October 25, 2017 at 11:59PM

Contents

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).

By the time you complete 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
```

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 to `staff@cs.tufts.edu`[1], 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` assignment, we expect you to use the initial basis, which is properly known as the Standard ML Basis Library[2]. 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]

[1] mailto:staff@cs.tufts.edu
[2] http://www.sml-family.org/Basis/
[3] http://sml-family.org/Basis/option.html#SIG:OPTION.option:TY
[4] http://sml-family.org/Basis/general.html#SIG:GENERAL.order:TY

- 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`

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If you have Jeff Ullman's text, you need to know that Chapter 9 describes the 1997 basis, which is out of date: today's compilers use the 2004 basis, which is a standard.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`[7].

Things you need to review before starting

We provide a guide to *Learning Standard ML*[8].

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 ML is not idiomatic. **Much of what you see in Ullman should not be imitated**. The code by Ramsey, 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.

I recommend that you focus on getting your code working first. Then submit it. Then pick up our "Style Guide for Standard ML Programmers"[9], 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[10].

Type checking versus unit testing

Standard ML is a language designed for production, not teaching. It has no `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
```

[5] http://sml-family.org/Basis/list.html#List:STR:SPEC
[6] http://sml-family.org/Basis/option.html#Option:STR:SPEC
[7] http://sml-family.org/Basis/text-io.html#SIG:TEXT_IO.inputLine:VAL
[8] ../readings/ml.pdf
[9] ../handouts/mlstyle.pdf
[10] ../handouts/mlstyle.pdf

You can also whip up a poor substitute for unit testing using this code as a model:

```
exception AssertionFailure
fun assert p = if p then () else raise AssertionFailure

val _ = assert(length ["a", "b", "c"] = 3)

val _ = assert(getOpt (NONE, 99) = 99)

val _ = assert(getOpt (SOME 12, 99) = 12)
```

Not great, but better than nothing.

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[11].

[11] ./cqs.ml.txt

1. Read section 5.1 of Harper[12] 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[13]. 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`:

   (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 365c. In evaluating `APPLY (f, args)`, if expression `f` does not evaluate to either a primitive function or a closure, the code raises the `RuntimeError` exception.

[12] http://www.cs.cmu.edu/~rwh/isml/book.pdf
[13] http://www.cs.cmu.edu/~rwh/isml/book.pdf

(a) Show a piece of µScheme code that would, when evaluated, cause chunk 365c to raise the `RuntimeError` exception.

→

(b) When exception `RuntimeError` is raised, what happens from the user's point of view?

You are now ready for problems G, L, and M.

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 Ramsey's book, which starts on page 375. Read the section and identify the free variables of the following expressions:

   (a) Free variables of `(lambda (x) (lambda (y) (equal? x y)))`

   →

   (b) Free variables of `(lambda (y) (equal? x y))`

   →

   You are now ready to improve the µScheme interpreter in problem 2.

Programming problems to solve individually (75%)

How to organize your code

Most of your solutions will go into a single file: `warmup.sml`. But in problem M, 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`[14] and `envfun-test.sml`[15].

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

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

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:

```
% /usr/sup/bin/mosmlc -c warmup.sml
% test-env-data
% test-env-fun
```

A previous version of this page said to run `test-env-code`; instead please run `test-env-fun`.

Please remember to **put your name and the time you spent in the `warmup.sml` file**.

The problems

Working on your own, please solve the following problems:

[14] ./envdata-test.sml
[15] ./envfun-test.sml

Defining functions using clauses and patterns

Related Reading for problems A to D: In *Learning Standard ML*[16] 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.

–>

C. Write 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`.

Lists

Related Reading for problems E to J: In *Learning Standard ML*[17], 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.

E. 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. 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`).

Do not use recursion in either part of this problem.

F. Implement `foldr` using recursion. Do not create unnecessary cons cells. Do not use `if`.

G. Write 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 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`.

H. Define a function

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

[16] ../readings/ml.html
[17] ../readings/ml.html

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

Write a version of `zip` which uses `pairfoldr` for its implementation. Call this function `zip2`.

I. Define a function

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

that turns a list of pairs into a pair of lists. Do not use `if`.

This one is tricky; here's a sample result:

```
- unzip [(1, true), (3, false)];
> val it = ([1, 3], [true, false]) : int list * bool list
```

You are welcome to translate any of the solutions from μScheme, but you may not use `if`.

Higher-order programming

Related Reading for problem K: The reading for the *Lists* problems should guide you in this section as well.

K. Function `compound` is something like a `fold`, but it operates on a restricted class of functions: the first argument to `compound` is a function of type `'a * 'a -> 'a`, which means it takes two arguments of the *same* type and returns a result also of that type. Examples of such functions include functions like `op +` and `op *`, but not `op ::` (cons). Every function that can be used with `compound` can be used with `foldr`, but not vice versa.

Function `compound` has this type:

```
val compound : ('a * 'a -> 'a) -> int -> 'a -> 'a
```

and `compound f n x` is

- `x` if `n=0`,
- `f (x, x)` if `n = 1`
- `f (x, f(x, x))` if `n = 2`
- in general, `f(x, f(x, f(x, ..., f(x, x))))`, where `f` is applied exactly `n` times.

Function `compound f` need not behave well when applied to a negative integer.

1. Write algebraic laws for `compound`. Use as few base cases as possible.
2. Implement `compound`.
3. Use your `compound` function to define a Curried function for integer exponentiation

   ```
   val exp : int -> int -> int
   ```

   so that, for example, `exp 3 2` evaluates to 9.

If you want to test `compound` with any of the predefined infix operators, you will need to convert the infix name to "nonfix" in an expression like

```
compound (op +) 10 1
```

Don't get confused by infix vs nonfix operators. Remember this:

- Fixity is a property of an identifier, not of a value.
- If `<$>` is an infix identifier, then `x <$> y` is syntactic sugar for `<$>` applied to a pair containing `x` and `y`, which can also be written as `op <$> (x, y)`.

Exceptions

Related Reading for problems L and M: In *Learning Standard ML*[18], 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.

L. Write a (Curried) function

```
val nth : int -> 'a list -> 'a
```

to return the *n*th element of a list. (Number elements from 0.) If `nth` is given arguments on which it is not defined, raise a suitable exception. You may define one or more suitable exceptions or you may choose to use an appropriate one from the initial basis. (If you have doubts about what's appropriate, play it safe and define an exception of your own.)

I expect you to implement `nth` yourself and not simply call `List.nth`.

M. *Environments with exceptions*.

1. In file `envdata.sml`, write definitions of a type `'a env` and functions

   ```
   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.
   - 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

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

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

3. In file `envboth.sml`, write a function

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

[18] ../readings/ml.html

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 imperative 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
cat envfun.sml  envboth.sml | mosml -P full
```

4. Also in file `envboth.sml`, write a function

```
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 environment. 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
cat envfun.sml  envboth.sml | mosml -P full
```

Algebraic data types

Related Reading for problems N and O: In *Learning Standard ML*[19], read the section on datatypes—Data IV. Make sure you understand how to pattern match on constructed values.

N. *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
```

[19] ../readings/ml.html

```
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)
```

- Write a function

  ```
  val addelt : 'a * 'a set -> 'a set
  ```

  that adds an element to a set.

- Write a function

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

  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

  ```
  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 P In *Learning Standard ML*[20], read the section on datatypes—Data IV. Make sure you understand how to pattern match on constructed values.

P. 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

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

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

3. Define function

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

[20] ../readings/ml.html

which returns the value that the finger points at.

4. Define functions

```
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

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

   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.

Related Reading for Exercise 2: Ramsey, Section 5.11, which starts on page 375. Focus on the proof system for judgment $y \in \mathrm{fv}(e)$; it is provable exactly when `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. 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 382, 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

  ```
  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 375.

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

  ```
  /usr/sup/bin/mosmlc -c mlscheme-improved.sml
  ```

- The second part is to write a function that takes a pair consistent 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 381, 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 365b. You simply apply `improve` to the pair that is already there, so your improved interpreter looks like this:

  ```
  (* more alternatives for [[ev]] for \uscheme 365b *)
  | 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. Compile both versions of the µScheme interpreter using MLton, which is an optimizing, native-code compiler for Standard ML.

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

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

I have provided a script that you can use to measure the improvement. I also recommend that you compare the performance of the ML code with the performance of the C code in the course directory.

- `time run-exponential-arg-max 22 ./mlscheme`
- `time run-exponential-arg-max 22 ./mlscheme-improved`
- `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. You will be using **all** of these concepts on 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 have a use for `map`, `foldr`,foldl, `or`List.filter'.

  You might also have a use for these functions:

  ```
  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.

- 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 two extra-credit problems: **FIVES** and **VARARGS**.

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 than cannot be computed by an expression involving exactly five 5's?
- What is the largest prime number that can 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

```
('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:

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

- Use dynamic programming[21].
- 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 384.

[21] ../readings/dynamic.html

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 loading declarations like the following *after* loading your solution:

```
(* first declaration for sanity check *)
val compound : ('a * 'a -> 'a) -> int -> 'a -> 'a = compound
val exp : int -> int -> int = exp
val firstVowel : char list -> bool = firstVowel
val null : 'a list -> bool = null
val rev : 'a list -> 'a list = rev
val minlist : int list -> int = minlist
val foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b = foldr
val zip : 'a list * 'b list -> ('a * 'b) list = zip
val pairfoldr : ('a * 'b * 'c -> 'c) -> 'c -> 'a list * 'b list -> 'c = pairfoldr
val unzip : ('a * 'b) list -> 'a list * 'b list = unzip
val nth : int -> 'a list -> 'a = nth
val addelt : 'a * 'a set -> 'a set = addelt
val treeFoldr : ('a * 'b -> 'b) -> 'b -> 'a tree -> 'b = treeFoldr
val setFold : ('a * 'b -> 'b) -> 'b -> 'a set -> 'b = setFold
val singletonOf : 'a -> 'a flist = singletonOf
val atFinger : 'a flist -> 'a = atFinger
val fingerLeft  : 'a flist -> 'a flist = fingerLeft
val fingerRight : 'a flist -> 'a flist = fingerRight
val deleteLeft  : 'a flist -> 'a flist = deleteLeft
val deleteRight : 'a flist -> 'a flist = deleteRight
val insertLeft  : 'a * 'a flist -> 'a flist = insertLeft
val insertRight : 'a * 'a flist -> 'a flist = insertRight
val ffoldr : ('a * 'b -> 'b) -> 'b -> 'a flist -> 'b = ffoldr
(* last declaration for sanity check *)
```

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`. Making sure that every function has the right type is your job, not mine.

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 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, 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*[22].

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 done either of the extra-credit problems, submit them as `varargs.sml` or `fives.sml`.

In comments at the top of your `warmup.sml` file, please include your name, the names of any collaborators, and the number of hours you spent on the assignment.

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.

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`.

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[23].

[22] ../readings/ml.html#expressions-viii-parentheses
[23] ../handouts/mlstyle.pdf