COMP 105 Homework: Core ML

~~Due Friday, March 1, at 5:59 PM, **six hours before midnight**.~~
Deadline extended! Due Saturday, March 2, at 11:59 PM.

The purpose of this assignment is to help you get acclimated to programming in ML:

- 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 core ML.

You will be more effective if you are aware of the Standard ML Basis Library. Jeff Ullman's text (Chapter 9) describes the 1997 basis, but today's compilers use the 2004 basis, which is a standard. You will find a few differences in I/O, arrays, and elsewhere; the most salient difference is in `TextIO.inputLine`.

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.

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.

  Code using `hd` or `tl` will earn No Credit.
- *Do not define auxiliary functions at top level*. Use `local` or `let`. Problems defining auxiliary functions at top level will earn No Credit.
- *Do not use* `open`; if needed, use one-letter abbreviations for common structures. Code using `open` may earn No Credit for your entire assignment.
- **Do not use any imperative features unless the problem explicitly says it is OK**.

Other guidelines

Some useful list patterns include these patterns, to match lists of *exactly* 0, 1, 2, or 3 elements:

```
<patterns>= [D->]
[]
[x]
[x, y]
[a, b, c]
```

and also these patterns, which match lists of *at least* 0, 1, 2, or 3 elements:

```
<patterns>+= [<-D]
xs
x::xs
x1::x2::xs
a::b::c::xs
```

When using these patterns, remember that **function application has higher precedence than any infix operator!** This is as true in patterns as it is anywhere else.

Use the standard basis extensively. Moscow ML's `help "lib";` will tell you all about the library. And if you use

```
ledit mosml -P full
```

as your interactive top-level loop, it will automatically load almost everything you might want from the standard basis.

All the sample code we show you is gathered in one place online.

As you learn ML, this table may help you transfer your knowledge of μScheme:

μScheme	ML
`val`	`val`
`define`	`fun`
`lambda`	`fn`

Put all your solutions in one file: `warmup.sml`. (If separate files are easier, combine them with `cat`.) 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*:

```
% /usr/sup/bin/mosmlc -c warmup.sml
%
```

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

Proper ML style

Ullman provides a gentle introduction to ML, and 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 textbook by Ramsey is a better guide to what ML should look like. We also direct you to these resources:

- The course supplement to Ullman
- The voluminous Style Guide for Standard ML Programmers

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

A collection of problems to be solved individually (90%)

Working on your own, please solve the following problems:

Higher-order programming

A. Here's a function that is somewhat like `fold`, but it works on binary operators.
   1. Define a function

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

      that ``compounds'' a binary operator `rator` so that `compound rator n x` is x if `n=0`, `x rator x` if `n = 1`, and in general `x rator (x rator (... rator x))` where `rator` is applied exactly `n`

times. `compound rator` need not behave well when applied to negative integers.
2. Use the `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. *Hint: take note of the description of `op` in Ullman S5.4.4, page 165.*

Don't get confused by infix vs prefix 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)`.

My estimate of difficulty: Mostly easy. You might be troubled by an off-by-one error.

Patterns

B. 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.
   1. `([1, 2, 3], ("COMP", 105))`
   2. `(("COMP", 105), [1, 2, 3])`
   3. `([("COMP", 105)], (1, 2, 3))`
   4. `(["COMP", "105"], true)`
   5. `([true, false], 2.718281828)`

   (Put your answers into `warmup.sml` as comments.)

   My estimate of difficulty: Easy
C. Using patterns, write a recursive Fibonacci function that does not use `if`. Calling `fib n` should return the `n`th Fibonacci number, e.g,. `fib 0` is 0, `fib 1` is 1, `fib 4` is 3. **Taking exponential time is OK.**
   My estimate of difficulty: Easy
D. 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`. *Remember that the ML character syntax is `#"x"`, as decribed in Ullman, page 13.*
   My estimate of difficulty: Easy to medium
E. 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.
   My estimate of difficulty: Easy

Lists

F. `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. Your solution should work regardless of the representation of integers (e.g., it should not matter how many bits are used to represent integers). Your solution can fail (e.g., by `raise Match`) if given an empty list of integers. Use `foldl` or `foldr`.

   Do not use recursion in any of your solutions.

My estimate of difficulty: Medium

G. Implement `foldl` and `foldr` using recursion. Do not create unnecessary cons cells. Do not use `if`.

   My estimate of difficulty: Easy

H. 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.

   My estimate of difficulty: Easy

I. Define a function

```
val pairfoldr : ('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`.

   Use `pairfoldr` to implement `zip`.

   My estimate of difficulty: Easy

J. Define a function

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

   that turns a list of pairs into a pair of lists. This one is tricky; here's a sample result:

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

   Hint: Try defining an auxiliary function, using `let`, that takes one or more accumulating parameters. And be prepared to use `rev`.

   My estimate of difficulty: Medium

K. Define a function

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

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

```
<sample>+= [<-D->]
- flatten [[1], [2, 3, 4], [], [5, 6]];
> val it = [1, 2, 3, 4, 5, 6] : int list
```

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

   My estimate of difficulty: Easy

Exceptions

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`.

My estimate of difficulty: Easy

M. Environments

1. Define a type `'a env` and functions

   ```
   <sample>+= [<-D->]
   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 = (* ... *)
   ```

   Defines `bindVar`, `emptyEnv`, `env`, `lookup`, `NotFound` (links are to index).

   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.
2. Do the same, except make `type 'a env = string -> 'a`, and let

   ```
   <sample>+= [<-D->]
   fun lookup (name, rho) = rho name
   ```

   Defines `lookup` (links are to index).
3. Write a function

   ```
   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 imperative features, like sequencing (the semicolon). Don't use `if`.
4. 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.*

My estimate of difficulty: Medium, because of those pesky functions as values again

Algebraic data types

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

```
<sample>+= [<-D->]
datatype 'a tree = NODE of 'a tree * 'a * 'a tree
                 | LEAF
```

Defines `tree` (links are to index).

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

```
<sample>+= [<-D->]
datatype order = LESS | EQUAL | GREATER     (* do not include me in your code *)
```

Defines `order` (links are to index).

Because `order` is predefined, if you include it in your program, you will hide the predefined version (which is in the ainitial 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.,

<sample>+= [<-D->]
```
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
```

Defines insert (links are to index).

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,

<sample>+= [<-D->]
```
datatype 'a set = SET of ('a * 'a -> order) * 'a tree
fun nullset cmp = SET (cmp, LEAF)
```

Defines nullset, set (links are to index).

- ♦ 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.

My estimate of difficulty: Easy to Medium

O. Sequences with constant-time append.
Consider the following informal definition of a sequence, by cases:

A sequence is either empty, or it is a singleton sequence containing one element x, or it is one sequence followed by another sequence.

1. **Define an algebraic datatype** `'a seq` that corresponds to this definition of sequences. As always, your definition should contain one value constructor for each alternative.
2. Define a function `scons : 'a * 'a seq -> 'a seq` that adds a single element to the front of a sequence, **using constant time and space**.
3. Define a function `ssnoc : 'a * 'a seq -> 'a seq` that adds a single element to the *back* of a sequence, **using constant time and space**.
   Note: The order of arguments is the **opposite** of the order in which the results go in the data structure. That is, in `ssnoc (x, xs)`, `x` *follows* `xs`. (The arguments are the way they are so that ssnoc can be used with foldl and foldr.)

4. Define a function sappend : 'a seq * 'a seq -> 'a seq that appends two sequences, **using constant time and space**.
5. Define a function listOfSeq : 'a seq -> 'a list that converts a sequence into a list containing the same elements in the same order. **Your function must allocate only as much space as is needed to hold the result**.
6. **Without using explicit recursion**, define a function seqOfList : 'a list -> 'a seq that converts an ordinary list to a sequence containing the same elements in the same order.

An immutable, persistent alternative to linked lists

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
   ```

   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.

My estimate of difficulty: Hard

One problem you can do with a partner (10%)

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 help you get off to a good start. It will also give you a feel for the kinds of things compiler writers do.

In this problem we will follow up on the discussion in class about how a compiler doesn't need to store an entire environment in a closure; it only needs to store the free variables of its lambda expression. For the details you will want the handout on free variables. (The exercise is Exercise 2 from the handout, which is **not** the same as Exercise 2 from the book.)

You'll solve the problem 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 to file 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, and it implements the proof rules on page 224 of the handout.

  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 handout, on page 225, 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 216b. You simply apply `improve` to the pair that is already there, so your improved interpreter looks like this:

```
        (* more alternatives for ev ((mlscheme)) 216b *)
        | ev (LAMBDA l) = CLOSURE (improve (l, 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.

  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 code 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.
My estimate of difficulty: Understanding free variables is hard, but once you understand, the coding is easy.

Extra credit

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

FIVES

Recall the following problem from the Scheme homework:

> *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. Write a µScheme program to answer one or more of the following 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, since they're easy to get with `setFold`.) My solution is under 20 lines of code, but it makes heavy use of the `setFold`, `nullset`, `addelt`, and `pairfoldr` functions defined earlier.

Hints:

- In order to be able to use `Int.compare`, 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.
- 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. Do so by giving the name `...` (three dots) special significance when it appears as the last formal parameter in a lambda. For example:

```
-> (val f (lambda (x y ...)) (+ x (+ x (foldl + 0 ...)))
-> (f 1 2 3 4 5) ; inside f, rho = { x |-> 1, y |->, ... |-> '(3 4 5) }
15
```

In this example, it is an error for `f` to get fewer than two arguments. If `f` gets at least two arguments, any additional arguments are placed into an ordinary list, and the list is used to initialize the location of the formal parameteter associated with `...`.

1. Implement this new feature inside of `mlscheme.sml`. I recommend that you begin by changing the definition of `lambda` on page 187 to

   ```
   and lambda = name list * { varargs : bool } * exp
   ```

   The type system will tell you what other code you have to change. For the parser, you may find the following function useful:

   ```
   fun newLambda (formals, body) =
      case rev formals
        of "..." :: fs' => LAMBDA (rev fs', {varargs=true},  body)
         | _            => LAMBDA (formals, {varargs=false}, body)
   ```

   The type of this function is

   ```
   name list * exp -> name list * {varargs : bool} * exp;
   ```

   thus it is designed exactly for you to adapt old syntax to new syntax; you just drop it into the parser wherever `LAMBDA` was used.
2. As a complement to the varargs lambda, write a new `call` primitive such that

   ```
   (call f '(1 2 3))
   ```

   is equivalent to

   ```
   (f 1 2 3)
   ```

   Sadly, you won't be able to use `PRIMITIVE` for this; you'll have to invent a new kind of thing that has access to the internal `eval`.
3. Demonstrate these utilities by writing a higher-order function `cons-logger` that counts cons calls in a private variable. It should operate as follows:

   ```
   -> (val cl (cons-logger))
   -> (val log-cons (car cl))
   -> (val conses-logged (cdr cl))
   -> (conses-logged)
   0
   -> (log-cons f e1 e2 ... en) ; returns (f e1 e2 ... en), incrementing
                                ; private counter whenever cons is called
   -> (conses-logged)
   99  ; or whatever else is the number of times cons is called
       ; during the call to log-cons
   ```
4. Rewrite the APPLY-CLOSURE rule to account for the new abstract syntax and behavior. To help you, simplified LaTeX for the original rule is online.

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:

```
<sample>+= [<-D]
(* first declaration for sanity check *)
val compound : ('a * 'a -> 'a) -> int -> 'a -> 'a = compound
val exp : int -> int -> int = exp
val fib : int -> int = fib
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 foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b = foldl
```

```
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 flatten : 'a list list -> 'a list = flatten
val nth : int -> 'a list -> 'a = nth
val emptyEnv : 'a env = emptyEnv
val bindVar : string * 'a * 'a env -> 'a env = bindVar
val lookup  : string * 'a env -> 'a = lookup
val isBound : string * 'a env -> bool = isBound
val extendEnv : string list * 'a list * 'a env -> 'a env = extendEnv
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 scons : 'a * 'a seq -> 'a seq = scons
val ssnoc : 'a * 'a seq -> 'a seq = ssnoc
val sappend : 'a seq * 'a seq -> 'a seq = sappend
val listOfSeq : 'a seq -> 'a list = listOfSeq
val seqOfList : 'a list -> 'a seq = seqOfList
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 ffoldl : ('a * 'b -> 'b) -> 'b -> 'a flist -> 'b = ffoldl
val ffoldr : ('a * 'b -> 'b) -> 'b -> 'a flist -> 'b = ffoldr
(* last declaration for sanity check *)
```

Defines `addelt`, `atFinger`, `bindVar`, `compound`, `deleteLeft`, `deleteRight`, `emptyEnv`, `exp`, `extendEnv`, `ffoldl`, `ffoldr`, `fib`, `fingerLeft`, `fingerRight`, `firstVowel`, `flatten`, `foldl`, `foldr`, `insertLeft`, `insertRight`, `isBound`, `listOfSeq`, `lookup`, `minlist`, `nth`, `null`, `pairfoldr`, `rev`, `sappend`, `scons`, `seqOfList`, `setFold`, `singletonOf`, `ssnoc`, `treeFoldr`, `unzip`, `zip` (links are to index).

I don't promise to have all the functions and their types here. Making sure that every function has the right type is **your** job, not mine.

Avoid common mistakes

Here is a list of common mistakes to avoid:

- 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, follow the directions in the course supplement to Ullman.

What to submit

There is no README file for this assignment.

- For your individual work, please submit the files `warmup.sml`, and optionally `varargs.sml` or `fives.sml`, using the script `submit105-ml-solo`. 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.
- For your joint work, 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 has changes, a few of the Form criteria are different.

More details will be delivered soon.

Index and cross-reference