Graduate Programming Languages, Assignment 5
Due: To Be Determined

Ensure you understand the course policies for assignments.

`hw5code.tar`, available on the course website, contains several Caml files you need.

For problems 1 and 2, do *not* use mutable references, mutexes, condition variables, fork-join, etc. All you need is thread-creation and the Concurrent ML primitives provided in the `Event` module. Remember to compile `cml.ml` via `ocamlc -vmthread -o hw5cml threads.cma cml.ml` or similar. Also note Caml terminates when the main thread terminates, so it is sometimes convenient when testing to block the main thread (e.g., by calling `Thread.yield` in an infinite loop).

1. (Concurrent ML) Implement the first commented-out interface in `cml.mli`. Define type `barrier` and functions `new_barrier` and `wait`. If a barrier is created by `new_barrier` i and a thread makes one of the first $i-1$ calls to `wait` with that barrier, then the thread should block until the i^{th} call, at which point all i threads should proceed. A thread that calls `wait` with a barrier after there have been i calls with that barrier should block forever.

   How to do it: `new_barrier` should return a channel that a newly created thread receives on. `wait` should send on this channel another channel and then receive on the channel it sends. (It does not matter what is sent on this channel; `()` is a fine choice.) The newly created thread should "remember" how many waiters there are and what channels to send on after the last arrives. The thread can then terminate. You do not need `choose` or `wrap`.

2. (Concurrent ML) Implement the second commented-out interface in `cml.mli`. This interface is for shared/exclusive locks (better known as readers/writer locks). Function `new_selock` creates a new lock. Functions `shared_do` and `exclusive_do` take a lock and a thunk and run the thunk. An implementation is correct if for each lock lk:

   - If there are any thunks that have not completed, then at least one of these thunks gets to run.
   - While a thunk passed to `exclusive_do` lk runs, no other thunk passed to `exclusive_do` lk or `shared_do` lk runs.
   - If there are no uncompleted exclusive thunks, then the uncompleted shared thunks get to run in parallel.

   (a) A correct solution to this problem has been given to you, but it is much longer and more complicated than necessary. In it, the "server" maintains explicit lists of waiting thunks. Reimplement the interface without these lists. Instead just use a simpler protocol that relies on the fact that CML allows multiple threads to block on the same channel, effectively forming an implicit queue. Hint: You still need `choose` and `wrap`, but you need fewer code cases and fewer total messages.

   (b) Suppose a call to `shared_do` lk blocks before a call to `exclusive_do` lk. For the two implementations of the interface (the one given to you and your simpler one), describe the states under which we are *sure* the call to `shared_do` lk will unblock before the call to `exclusive_do` lk. Explain your answer, which can be different for the two implementations. Note `choose` is nondeterministic.

3. (Implementing Class-Based OOP) You will complete the code in `oo.ml`, which demonstrates one way to implement simple object-oriented programming. There are *two* similar languages involved: the "source" language and the language that the interpreter expects.

   The "source" language is an ML expression of type `((classname option) classdef list) * exp` i.e., a pair of (1) a list of class definitions where the `super` field contains a `classname option` and (2) a "main" expression. The `super` field is the immediate superclass if there is one. You may *assume* the following: class names are distinct, the superclass relation has no cycles, all superclasses are defined in the program, the methods in one class definition have distinct names, the fields in one class definition have names distinct from each other and from all fields in all superclasses. A subclass extends its superclass by adding all its fields and any methods with names different from those in any superclass. To override a method, a class definition just includes a method of the same name (see the challenge problems).

   The language for our interpreter is an ML expression of type `((classname list) classdef list) * exp`. Unlike the source language:

   - The `super` field holds a list of all superclasses (transitively). There may be none.
   - The `methods` field holds all the methods for a class, including those defined in superclasses.
   - The `fields` field holds all the fields for a class, including those defined in superclasses.

   (a) The code given to you defines a `boolclass` in the source language. Define two subclasses `trueclass` and `falseclass`. Each should override the `ifThenElse` method, using the encoding of true returning the first of two objects and false the second.

   (b) Implement `translate_classdefs` of type `(classname option) classdef list -> (classname list) classdef list` for translating a list of class definitions from the source language to the interpreter language. The sample solution uses several helper functions and several functions in Caml's list library, and is 30–35 lines.

   (c) Complete the interpreter `interp` of type `classname list classdef list -> (varname * valu) list -> valu -> exp -> valu`. Three cases are done for you. The other cases are described informally below; there should be no surprises. In general, you may throw whatever sort of exception you like for bad programs, even `List.Not_found`. The arguments to `interp` are the (translated) class definitions (needed for method calls), an environment (needed for local variables), the value currently bound to self, and the expression being evaluated. Note `New` creates a `valu` where the fields start uninitialized.

   - Getting a field fails if the result of evaluating the subexpression does not have that field or that field is still uninitialized. Otherwise, it returns the `valu` held in the field.
   - Setting a field fails if the result of evaluating the first subexpression does not have that field. Otherwise, it does the appropriate mutation and returns the field's new contents.
   - A let-expression is interpreted very much like in ML.
   - A cast fails if the class of the result of evaluating the subexpression is neither the class specified nor a (transitive) subclass of it. Otherwise, the result is the result of the subexpression.
   - A local variable is looked up in the environment.
   - A method call evaluates all the subexpressions and then looks up the method in the class of the receiver. The lookup just uses the name of the method, returning the first method in the list with that name. The result is the evaluation of the method body using the receiver for self and an environment containing bindings just for the method arguments. It is an error to have the wrong number of arguments.

   (d) In a short English paragraph, explain why having casts in our language is of little use. In particular, give *two* standard uses of casts in languages like Java and C++, one of which affects type-checking and one of which affects method-call semantics. Explain how neither is relevant with our implementation.

(e) **(Challenge Problems)**

- (Fairly easy) Change the interpreter to support "arity overloading" — a method call should not fail due to having the wrong number of arguments if a superclass of the receiver defined a method with the same name and the correct number of arguments.
- (Fairly easy) Change your interpreter to detect type mismatches on field-assignment or method-call and fail immediately.
- (Much more challenging) Changing your translation but not your interpreter, implement static overloading. Document ambiguities and how you deal with them. (Small hints: `Fail` causes ambiguities. You will need a type-checker even though the language is not type-safe.)

Turn in: Written/typed answers to 2b and 3d, and Caml code in files `oo.ml` and `cml.ml`