1. Convert the following ML functions into continuation-passing style, i.e., so that all calls are tail calls. You may not use any imperative features. Submit your solutions to both the exercises in a single file `sol8_1.sml`.

   a. `fun fact n = if n < 2 then 1 else n * fact(n-1)`
   b. `fun fib n = if n < 2 then 1 else fib(n-1) + fib(n-2)`

2. Consider the following object-oriented language, which we’ll call “E8.”

   ```
   prog := '(' '{ class } ')
   class := '(' cname cname '{ member } ')
   member := '(' 'field' var exp ')
           | '(' 'method' mname '{ { var } ' exp ')
   exp := var
       | int
       | '(' ':=' var exp ')
       | '(' 'while' exp exp ')
       | '(' 'if' exp exp exp ')
       | '(' 'write' exp ')
       | '(' 'block' '{ exp } ')
       | '(' 'local' '{ { var exp } ' exp ')
       | '(' '0' exp mname '{ exp } '
       | '(' 'new' cname ')
       | '(' '+' exp exp ')
       | '(' '-' exp exp ')
       | '(' '*' exp exp ')
       | '(' '/' exp exp ')
       | '(' '<=' exp exp ')
   var := letter { letter | digit }
   mname := letter { letter | digit }
   cname := letter { letter | digit }
   ```

   As usual, comments may be included by enclosing them between '{' and '}' characters, and they may be nested.

   E8 is similar to the language E3, but with object-oriented features added. Instead of top-level function and global definitions, there are top-level class definitions. Class definitions consist of a class name, a superclass name, and a list of members. A member is either a field or a method. A field has an identifier and an initializing expression. A method has a name, zero or more parameter names, and a body expression. Expressions are much as before,
except that the format of applications has changed to specify an object expression and a
method name from that object to be applied, and there is a new constructor expression to
create objects of a specified class.

The semantics of E8 are similar to those of other class-based object-oriented languages such
as Java or Smalltalk. Classes are arranged in a hierarchy, with an (empty) built-in class
Object at the root. Objects are created (via new) as members of a named class. A newly
created object has all the members of that class, plus the members of all its ancestors.

The initial value of each field is specified by an expression in the field definition. Fields can
contain integers or other objects. The intention is that fields of a object are only accessible for
reading and writing within functions defined within the class of that object or its subclasses.
(Actually, they are also accessible from functions defined in its superclasses, but that is really
a kind of bug in my implementation; if this were a typed language, such accesses could easily
be prohibited statically.)

All function applications are “virtual” (in C++ terminology); i.e., when a function is applied,
itself is searched for first in the receiving class, then in its superclass, and so on. Within
any function body, the identifier “this” is always pre-defined to refer to the receiving object
of the function call.

As usual, further details of the semantics may be revealed by experimenting with the (Java)
interpreter (hw8_2.java) and reading its source code!

Your task is to modify the interpreter in two ways. Combine your modifications into a single
file sol8_2.java and submit this file.

(a) Add a new expression form

\[
\text{exp := } \ldots
\]

\[
| (\text{’}’ \text{’} \text{instanceof} \text{’} \text{’} \text{cname} \text{’} \text{exp} \text{’})
\]

The expression (instanceof c e) should return the integer 1 if expression e evaluates to
an object which was created by (new c') where c' is c or a subclass of c; otherwise it should
return 0. Using instanceof as the test expression of an if gives a way to branch according
to the runtime class of an object. (This is essentially the same as the instanceof operator
in Java.)

Note: You’ll need to add your own parsing support for instanceof; by now, this should be
trivial for you to figure out.

(b) Modify the method inheritance mechanism as follows. Suppose we invoke a method m
on an object x of class C. To find the definition of m, we search up the class hierarchy starting
from class C, as usual. If m is defined directly in C, all works just as now. But if m is found
in some superclass D of C, then any method lookups on x that occur during the execution
of m should start with class D rather than with the original class C. If done properly, this
change will break code like the translate_and_draw method described in lecture. Include a
comment in your code indicating why that sort of method now fails.

Hint: You can implement the new lookup behavior by temporarily coercing x to a D while m
is executing. This can be done simply by temporarily changing the cname field.