CSE 505, Fall 2005, Assignment 4
Due: Tuesday 29 November 2005, 5:00PM

`hw4.tar`, available on the course website, contains several Caml files you will need.
Last updated: November 14
Advice: For problem 1, be warned that the "intellectual difficulty" is high, but the amount of code required is low (sample solution adds 25 lines to `main.ml` and 7 lines to `ast.ml`). Do not hesitate to ask for help.

1. (Machines; Continuations) You are given an untyped-λ-calculus implementation that uses an "explicit stack and environment" machine that is like the last machine considered in lecture 10. The interpreter uses one tail-recursive function (i.e., a loop). On each iteration, we have the "current expression" to be evaluated, the "environment" to evaluate variables and a "context stack" that encodes what to do after the current expression. Note that when (1) creating a closure or (2) pushing an "expression to be evaluated later" onto the stack, we store the current environment so that we can use that environment later when (1) evaluating the function body or (2) evaluating the expression. (In lecture 10, we got (1) right but (2) wrong; Dan has updated the lecture 10 materials to fix this bug.)

   (a) Extend the machine to support pairs and conditionals (i.e., every expression and value form defined except `Letcc`, `Throw`, and `Continuation`). You must maintain tail-recursion. Hints:
   - Add 5 constructors to `Ast.context_elt`, each "carrying" somewhere between 0 and 3 things.
   - Add 4 cases to the outer match in `loop` and 5 cases to the inner match in `loop` (3 of which need inner-inner matches).
   - Raise `RunTimeError` if the program reads a field of a non-pair or branches on a non-boolean.
   - Note the abstract syntax distinguishes between "pair builders" `MP(e1,e2)` and "pair values" `P(v1,v2)`. So the last step of evaluating `MP(e1,e2)` is to build an expression of the form `V(P(v1,v2))` (and the next iteration will then use the stack to "see what to do next").

   (b) Extend the machine to support (first-class) *continuations*. You must maintain tail-recursion. To understand continuations, recall our evaluation-context semantics for λ-calculus:

$$E ::= [\cdot] \mid E\ e \mid v\ E \mid (E,e) \mid (v,E) \mid \mathsf{if}\ E\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \qquad \frac{e \xrightarrow{\mathrm{p}} e'}{E[e] \to E[e']}$$

$$\frac{}{(\lambda x.\ e)\ v \xrightarrow{\mathrm{p}} e[v/x]} \qquad \frac{}{(v_1,v_2).1 \xrightarrow{\mathrm{p}} v_1} \qquad \frac{}{(v_1,v_2).2 \xrightarrow{\mathrm{p}} v_2}$$

$$\frac{}{\mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 \xrightarrow{\mathrm{p}} e_1} \qquad \frac{}{\mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 \xrightarrow{\mathrm{p}} e_2}$$

   A continuation represents "what to do for the rest of the program." We can "grab the current continuation" with letcc and "change the current continuation" with throw:

$$\begin{array}{rcl} e & ::= & \ldots \mid \mathsf{letcc}\ x.\ e \mid \mathsf{throw}\ e\ e \mid \mathsf{continuation}\ E \\ v & ::= & \ldots \mid \mathsf{continuation}\ E \\ E & ::= & \ldots \mid \mathsf{throw}\ E\ e \mid \mathsf{throw}\ v\ E \end{array} \qquad \frac{}{E[\mathsf{letcc}\ x.\ e] \to E[e[\mathsf{continuation}\ E/x]]}$$

$$\frac{}{E[\mathsf{throw}\ (\mathsf{continuation}\ E')\ v] \to E'[v]}$$

   Note the two new semantic rules use $\to$ not $\xrightarrow{\mathrm{p}}$. In English, letcc binds to x a "continuation value" that stores the context that was current when the letcc was evaluated. (If x does not appear free in e, then letcc $x.\ e$ and e are equivalent.) When we throw to a continuation with E' stored, E' becomes the current context (the "old E" disappears!). We are stuck if the first argument to throw does not evaluate to a continuation. Hints:
   - Add 2 constructors to `Ast.context_elt`, each "carrying" 1 or 2 things.
   - Add 2 cases to the outer match in `loop` and 2 or 3 cases to the inner match in `loop` (depending on how you detect a non-continuation as the first argument to throw).
   - Raise `RunTimeError` if the program tries to throw to a non-continuation.

(c) Continuations are very powerful, but in this problem we use them for something basic: halting a program. Change `Main.allow_halt` such that:

- It takes an expression e and returns an expression e'.
- e can have free occurrences of the variable `halt` and call it as a *function* taking one argument, i.e., `halt` e''.
- If e evaluates to v without ever calling `halt`, then e' evaluates to (true, v).
- If e evaluates after some number of steps to $E[\texttt{halt } v]$, then e' evaluates to (false, v).
- e' contains e as a subexpression – that is, do *not* look at e, just wrap it with some outer code.

Sample solution is 3 lines. Put e in a function that takes `halt` as an argument. Pass this function a function that contains a `throw`. Advice: Work out your solution on paper first.

2. (Subtyping) Implement a subtype-checker using `q2.ml`. That is, implement the `is_subtype` function to return `true` if and only if the first "type" it is passed (as defined by `atype`) is a subtype of the second, given the mapping from type-names to types in `ctxt`. You may assume there are no recursive types (see the extra credit), meaning no `typecontext` will ever map a name to a type that (transitively) contains the name (even via more name-lookups). Your algorithm should support these notions of subtyping:

   - Named types are equivalent to their definitions. So a named type `t1` is a subtype of `t2` if the name's definition is a subtype of `t2`, and a type `t3` if a subtype of a named type `t4` if `t3` is a subtype of `t4`'s definition.
   - `Int` is a subtype of `Int`.
   - Function types have their usual contravariant-argument covariant-result subtyping.
   - Tuples have width and depth subtyping.
   - Sums have "anti"-width (supertype can have more constructors), depth (supertype can have same constructor carrying a supertype), and permutation (order of constructors does not matter) subtyping.

   Notes:

   - Sample solution is 23 lines. Matching on the pair `(t1,t2)` helps keep things concise. You can make `is_subtype` recursive, but since `ctxt` does not change, it's more concise to make a recursive helper function.
   - If a named-type appears that is not in the context, raise the `NoNamedType` exception.
   - You can use the provided `lookup` function for contexts and the lists with sum types; for the latter, the caller should use a `try` expression.

3. (Polymorphic Types) Suppose we extend System F with a polymorphic `let rec` where `let rec` $\alpha_1, \ldots, \alpha_n$ `f x =` e creates a recursive function with type parameters $\alpha_1, \ldots, \alpha_n$ and argument `x`.

   (a) Give the appropriate typing rule for `let rec` $\alpha_1, \ldots, \alpha_n$ `f x =` e.

   (b) Give a full typing derivation showing that this program can have type $\forall\alpha_1\forall\alpha_2.\alpha_1 \to \alpha_2$:
   `let rec` α_1, α_2 `f x = f`$[\alpha_1][\alpha_2]$ `x`.

   (c) Suppose we have exceptions and `raise` has type `exn`$\to \alpha$ as in Caml and that `Foo` has type `exn`. Give a full typing derivation showing that this program can have type $\forall\alpha_1\forall\alpha_2.\alpha_1 \to \alpha_2$:
   $\Lambda\alpha_1.\ \Lambda\alpha_2.\ \lambda x.$ `raise Foo`.

   (d) In general, suppose v is a Caml value of type $\forall\alpha_1 ... \forall\alpha_n.\tau \to \alpha$ and α is not free in τ. In English, what can you say about v?

4. (Protocol Enforcement) Note: A skeleton for your solution is in `q4.ml`. Also note `open` is a Caml keyword. You need to add only a few lines of code.

   Suppose there are two functions, `openup` of type `unit->unit` and `read` of type `unit->string`, and it is an error to call `read` before calling `openup`.

   (a) `client1` takes functions of type `unit->unit` and `unit->string` but cannot be trusted to obey the protocol. Write `dynamicCheck` which should call `client1` with functions that raise the `ProtocolViolation` exception as soon as the protocol is violated else do the right thing.

   (b) `client2` has a different interface in which it is only passed one function, which returns a second function when called. Write `withholdRead` which should call `client2` with a function that when called "does the open" and "provides the ability to read."

   (c) `client3` has a different interface that is "more polymorphic." Write `useTypes` which calls `client3` with the functions it is passed.

   (d) In English, explain why clients 2 and 3 cannot violate the protocol and why they are more efficient than client 1.

   (e) Suppose our protocol also has a `close` function of type `unit->unit` and it is an error to call `read` after `close`. In English, explain which of the 3 approaches above will work in this setting and why the other approaches do not work. (Note: There are language features and type systems that do this sort of thing well, but we won't have to time to investigate them.)

5. (**Extra Credit**)

   (a) Use your implementation of continuations to support cooperative multithreading (warning: Dan has not done this, but it's a classic idea):

   - Extend the syntax to support spawn(e) and yield.
   - Write a source-to-source transformation that removes all uses of spawn and yield, i.e., do *not* change `interp`.
   - For simplicity, assume the input program does not use the variables `_q` or `_ans`. Change the input program so that every function takes two more arguments `_q` and `_ans` (use currying) and every function application passes on its `_q` and `_ans`. (We are passing "global variables" through the program.) Wrap the whole program in a function that takes `_q`, `_ans`, `spawn`, and `yield` and returns `_ans`.
   - `_q` is the list (the queue) of non-running threads. `_ans` is the list of values produced by threads. (Use pairs and booleans to encode lists.) Queue elements are continuations (that ignore the value they are thrown). So `_q` and `_ans` start empty. `spawn` is just a function that adds to the queue. `yield` is a function that ignores its argument, stores the running thread on the queue, and starts the next waiting thread. (It uses letcc to "get what to store" and throw to "start the next thread.")
   - When a thread terminates (i.e., the argument to spawn finishes), it adds its final value to the front of `_ans`. When the last thread terminates, it adds its value to `_ans` and returns `_ans`.
   - Threading is a bit silly in a language with no inter-thread communication; without `_ans` spawning threads would have no effect on a program's answer.
   - For writing test programs, note that you can encode $e_1; e_2$ with $(\lambda x.\ e_2)e_1$ provided x does not occur free in e_2.

   (b) Extend your datatype-subtype checker to support recursive datatypes.

What to turn in:

- Caml source code in `main.ml`, `q2.ml`, and `q4.ml`. Put extra credit in different, suitably named files.
- Hard-copy answers to problems 3, 4d, and 4e.

Email your source code to Erika as `firstname-lastname--hw4.tgz` or `firstname-lastname--hw4.zip`. Hard copy solutions should be put in Erika's grad student mailbox, in the envelope outside her office, or given to her directly.