CSE P505, Spring 2006, Assignment 3
Due: Tuesday 9 May 2006, 5:00PM

Last updated: April 24. Except for problem 1, consult `hw3.ml`. Problem 1 is independent of other problems.

1. (Formal Semantics)

   (a) Give a formal large-step operational semantics for the Logo language from homework 2, but where a program just produces a position and direction (not a trace). Here is the BNF for Logo with a new line for *pos*. (A *pos* is three numbers x, y, and d for the current position and direction.)

$$e ::= \mathsf{home} \mid \mathsf{forward}\ f \mid \mathsf{turn}\ f \mid \mathsf{for}\ i\ lst$$
$$lst ::= [] \mid e::lst$$
$$pos ::= (x, y, d)$$

   Requirements:

   - Your judgment should have the form $(pos, lst) \Downarrow pos'$, meaning "Starting from *pos*, the move list *lst* ends in pos'."
   - Your inference rules should be consistent with this partial derivation (i.e., the tree can be built by instantiating two of the rules):

$$\dfrac{\dfrac{((0,0,0), \mathsf{forward}\ 1::\mathsf{turn}\ \pi::[]) \Downarrow (1,0,\pi) \qquad ((1,0,\pi), (\mathsf{for}\ 1(\mathsf{forward}\ 1::\mathsf{turn}\ \pi::[]))::[]) \Downarrow (0,0,0) \qquad 2 > 0}{((0,0,0), (\mathsf{for}\ 2(\mathsf{forward}\ 1::\mathsf{turn}\ \pi::[]))::[]) \Downarrow (0,0,0)}}{(pos, \mathsf{home}::(\mathsf{for}\ 2(\mathsf{forward}\ 1::\mathsf{turn}\ \pi::[]))::[]) \Downarrow (0,0,0)}$$

   Assume the metalanguage has all arithmetic and list operations you need.

   Begin Hints

   - This is a hand-written or typed problem; see the turn-in instructions.
   - All you need to do is write 6 inference rules: 1 for the empty move-list and 1 for each possible "first move" (with two rules for loops since i might be 0). Produce the tree above by instantiating the "home" rule and the "loop > 0 times" rule.
   - The only rule that needs more than 1 premise is the "loop > 0 times" rule. This is the trickiest rule: have 3 premises, two of which are "recursive" (unroll the loop) and one of which is just $i > 0$.
   - Metalanguage operations you need are arithmetic, trigonometry, mod, and list cons (::).
   - Looking ahead to 1(b) may help. Be sure your solutions to 1(a) and 1(b) end up consistent.

   End Hints

   (b) Give a full derivation tree showing that for all *pos*:

$$(pos, \mathsf{home}::(\mathsf{for}\ 2\ (\mathsf{forward}\ 1::\mathsf{turn}\ \pi::[]))::[]) \Downarrow (0,0,0)$$

   All you need to do is expand the partial tree above to give derivations for the two "recursive" premises.

   Begin Hints

   - As with any hand-evaluation, this is tedious and error-prone, but it's unclear how else to understand instantiating inference rules.
   - Do metalanguage operations implicity. For example, if a rule has $x + r\cos d$ in it and you're instantiating the rule with $x = 0$, $r = 1$, $d = 0$, then just write 1 instead of $0 + 1\cos 0$.
   - The sample solution has 11 horizontal lines including the 2 given to you and including 3 instantiations of axioms (i.e., no premises).
   - Give your inference rules names and label each step in your tree with the name so it's easier for you and the grader to see what you are instantiating.

   End Hints

2. (Lambda-Calculus) Complete function `interp` to support the larger lambda-calculus defined by `exp` in `hw3.ml`. This is the environment-based semantics from lecture 4, but you must add support for:

   - integer constants (which are values)
   - addition (sums the results of its subexpressions)
   - conditionals (1st expression should evaluate to an integer (some `Int i`); 2nd and 3rd expressions can be *any* 2 expressions. Evaluates the 3rd expression if integer (`i`) is 0 else the 2nd expression)
   - pairs (evaluates its subexpressions; a pair of values is a value)
   - pair accessors (return a pair-component; `First` and `Second` have their expected meaning)

   Using `Plus`, `If`, `First`, or `Second` on the wrong sort of value(s) must raise `RuntimeTypeError`.
   Begin Hints

   - Do not change the definition of `exp` or any of the cases provided to you.
   - The `f` argument to `interp` and third part of `Lam` is for the next problem so we can change how we create closures' environments (see the `Closure` case) without copying the interpreter. For this problem, just imagine `f` is `fun x _ -> x` (as in `interp1`) and the third part of a `Lam` is ignored.

   End Hints

3. (Slimmer Environments) Complete function `computeFreeVars` to have type `exp -> exp * string list`. The `string list` result is the *free variables* in the argument. The `exp` result is like the argument except for every `Lam`, the `string list option` is now `Some lst` where `lst` contains the function's free variables (and has no duplicates).

   Begin Hints

   - `computeFreeVars` is a "preprocessing" step. The result can be evaluated with `interp2` (written for you). This interpreter stores with closures a "filtered" environment that contains only variables occurring free in the function.
   - Use the helper functions at the top of `hw3.ml` for treating a list of strings as a set.
   - You are making a "deep copy" of the expression.
   - You may/should assume the argument is a "source" program and therefore has no `Closure` expressions. Raise `BadSourceProgram` if you encounter one.

   End Hints

4. (Comparing efficiency) Consider `interp1` and `interp2` and assume (falsely given this implementation, though it's not hard to do properly) that variable lookup in environments takes O(1) time. Describe the following (using just a few English sentences for each):

   (a) Programs where `interp1` would require significantly *more space* for evaluation than `interp2`.

   (b) Programs where `interp1` would require significantly *less space* for evaluation than `interp2`.

   (c) Programs where `interp1` would require significantly *more time* for evaluation than `interp2`.

   (d) Programs where `interp1` would require significantly *less time* for evaluation than `interp2`.

5. (A third approach)

   (a) Explain in a few English sentences the semantics of `interp3` and why this is appears to be a *much* less useful interpreter.

   (b) **EXTRA CREDIT** Explain (it will take more than a few sentences) how `translate` works so that the `exp` it produces can be evaluated correctly by `interp3`. (Hint: It's the key idea behind *closure conversion*.)

Turn in:

- Email your `hw3.ml` to Ben as an attachment.
- **If you are using Seminal, please include your backup files.**
- For problem 1, emailing an electronic copy (pdf preferred to ASCII preferred to Word but all are okay; scanned handwriting is also fine) is preferred. Faxing and hand-delivery are also options if necessary.
- For problems 4 and 5, a text document (ASCII and pdf are fine; Word is okay) is required.