CSE P505, Autumn 2016, Homework 3
Due: Wednesday 9 November 2016, 11:00PM

- This assignment covers formal semantics and interpreters for lambda-calculus-like languages. There is no type system (we will get there in the *next* homework).
- Instructions regarding `*.ml` files in this document apply to their `*.fs` counterparts except where noted.
- Problem 1 is independent of the other problems. You can type it or write it by hand and scan it / photgraph it / whatever.
- Problems 2 and 3 can be done just with `prob23.ml`, but see next point.
- Problems 4 and 5 need `prob23.ml`, `prob23.mli`, and `prob45.ml` (F#: only `prob23.fs` and `prob45.fs`). We have also provided a `Makefile` and a `main.ml/fs`. **The easiest thing to do is to compile by running `make` but that is not required.** The sequence of commands in the `Makefile` creates a program that runs `prob23.ml`, then `prob45.ml`, then `main.ml`. The role of `.mli` files and how to access bindings from other modules were discussed in Lecture 2. (F#: To build `main.exe` from the command line, simply run `fsharpc prob23.fs prob45.fs main.fs`.)
- Turn in your solution via the "Turn-in" link on the course website. Include `prob23.ml`, `prob45.ml`, and a third file with your solutions to problem 1 and the English parts of problem 3. Do not modify the other code files.
- Understand the course policies on academic integrity (see the syllabus) and challenge 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 (final) 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.

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

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

$$
\dfrac{\dfrac{\dfrac{\vdots}{((0,0,0), \mathsf{forward}\ 1{::}\mathsf{turn}\ \pi{::}[]) \Downarrow (1,0,\pi)} \quad \dfrac{\vdots}{((1,0,\pi), (\mathsf{for}\ 1(\mathsf{forward}\ 1{::}\mathsf{turn}\ \pi{::}[])){::}[]) \Downarrow (0,0,0)} \quad 2 > 0}{((0,0,0), (\mathsf{for}\ 2(\mathsf{forward}\ 1{::}\mathsf{turn}\ \pi{::}[])){::}[]) \Downarrow (0,0,0)}}{((17,19,4), \mathsf{home}{::}(\mathsf{for}\ 2(\mathsf{forward}\ 1{::}\mathsf{turn}\ \pi{::}[])){::}[]) \Downarrow (0,0,0)}
$$

   Hints:

   - All you need to do is write 6 inference rules.
   - 1 rule is for the empty move-list.
   - 2 rules are for when the first move in a move-list is a loop.

2. (Lambda-Calculus) Complete function `interp` to support the larger lambda-calculus defined by `exp` in `prob23.ml`. This is the environment-based semantics from Lecture 5. It is a large-step interpreter, so the result should be a value. You need to add support for:

   - integer constants, which are values
   - addition: sums the results of its subexpressions
   - boolean constants, which are values
   - test-for-zero: evaluate its argument to an integer constant and return `True` if the constant is 0 else `False`.
   - conditionals: first expression should evaluate to `True` or `False`; second and third expressions can be *any* 2 expressions. Evaluates the second expression if first evaluates to `True`, evaluates the third expression if first evaluates to `False`, else it is an error.
   - pairs: evaluates its subexpressions. A pair of values is a value.
   - pair accessors: `First` and `Second` have their expected meaning.

   Using `Plus`, `Iszero`, `If`, `First`, or `Second` on the wrong sort of value(s) must raise `RuntimeTypeError`. In particular, it is an error to use `Iszero` with a subexpression that does not produce an integer.

   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.
   - Sample solution is less than 40 lines, including the cases provided to you.
   - Beware nested match errors: If you use a match expression inside another match expression, parentheses around the inner one is a good idea so that the compiler does not think the next pattern of the outer match belongs to the inner match. For example, the parentheses around the match expression in the `Apply` case provided to you are necessary. (F#'s whitespace-is-relevant syntax makes the parentheses unnecessary.)

3. (Slimmer Environments)

   (a) Complete function `computeFreeVars` to have type `exp -> exp * string list`, which is the same as `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). Recall free variables were discussed in Lecture 4. They are the variables that occur in an expression somewhere that they are not bound. (Presumably a containing expression binds them.)

   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 `prob23.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.
   - Sample solution is less than 40 lines.

(b) 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):

i. Programs where `interp1` would require significantly *more space* for evaluation than `interp2`.

ii. Programs where `interp1` would require significantly *less space* for evaluation than `interp2`.

iii. Programs where `interp1` would require significantly *more time* for evaluation than `interp2`.

iv. Programs where `interp1` would require significantly *less time* for evaluation than `interp2`.

4. (Translating away sums) Complete function `translate` in file `prob45.ml`, which has type `exp -> Prob23.exp`. The source language for this translation is a lambda-calculus that is like the target language except it does not have conditionals or booleans and it does have sums. (By sums, we mean constructors and a match expression, *not* addition, though the language has that too.) The sums have the same meaning as discussed in Lecture 6. The file contains an example use of the `Match` constructor. The meaning of `Match(e1,s2,e2,s3,e3)` is: If `e1` evaluates to `A v`, then the result is the result of `e2` evaluated under the environment extended to map `s2` to `v`. Else if `e1` evaluates to `B v`, then the result is the result of `e3` evaluated under the environment extended to map `s3` to `v`. Else it is an error.

   The result of `translate` should be an expression that when interpreted by `Prob23.interp1` produces an appropriate answer. In other words, the translation should be correct. In particular, if a program should evaluate to an integer `Int i`, then it should evaluate to `Prob23.Int i`. Your translation should not evaluate a subexpression more times than necessary (see the third hint).

   If the source program should raise an error, it is okay if the translation does not. An interesting not-required exercise is to come up with such a program.

   Hints:

   - You will need to translate the entire program since subexpressions of a `Prob23.exp` must also have type `Prob23.exp`.
   - The interesting cases are the three cases related to sums. You will need to design these cases together, coming up with a translation of sums for which your translation of match works correctly. Use pairs and booleans, similar to what we said in lecture was "bad style." (Here we have no choice and style is less of an issue for the result of an automatic translation.)
   - **You may assume that the variable `"__tmp1"` does not appear in the source program.** This small "cheat" makes it a little easier to create a translation that doesn't evaluate the first subexpression in a match more than once.[1]
   - The sample solution is less than 20 lines and uses the `makeLet` helper function provided to you multiple times in one case.

5. (Continuation-Passing Style) Complete function `translate2_cps` in `prob45.ml` so that its result is always the same as `translate` but all recursive calls to translate subexpressions are tail-calls. Note it is the code in `translate2_cps` that you should write in continuation-passing style; the lambda-calculus program you *produce* should still be in direct style (the same program that `translate` would produce).

6. **Challenge problem:** (A third approach)

   (a) Explain in a few English sentences the semantics of `interp3` in `prob23.ml` and why this appears to be a much less useful interpreter.

   (b) Explain (it will take more than a few sentences) how the function `translate` in `prob23.ml` works so that the `exp` it produces can be evaluated correctly by `interp3`. Hint: It's the key idea behind *closure conversion*.

[1] Without this "cheat" we would risk capture. The proper way to do this would be to look at the program and find a variable not being used rather than making up an obscure variable name.