CSE P505, Autumn 2016, Homework 2
Due: Wednesday October 26, 11:00PM

- This assignment emphasizes implementing programming languages by interpretation or translation.
- Understand the course policies on academic integrity (see the syllabus) and challenge problems.
- Modify `imp.ml` and `logo.ml`, available on the course website, to produce your solution.
- Do *not* use mutation.
- To turn in your solution, follow the "Turn-in" link on the course website.
- If you do the challenge problems, put them in `imp2.ml` or `logo2.ml` as appropriate.
- The language for problem 2 is described in a separate document, `logo.pdf`, on the course website.

1. Modify the large-step IMP interpreter (provided in lecture 2; see `imp.ml`) to support saving and restoring entire heaps. In particular, implement two new statement forms:

   - saveheap *str* is like assignment except instead of putting an integer in *str*, it puts the (entire) current heap.
   - restoreheap *str* takes the heap stored in *str* and makes it the current heap.

   A heap variable can store an int or a heap, so we need these "cheating rules":

   - If x holds a heap, then the expression x evaluates to 0.
   - If x holds an int, then the statement restoreheap x has no effect (it is like a skip).

   Hints:

   - So that heap elements can hold integers or other heaps, change the type `heap` to be mutually recursive with a new datatype you define. To define mutually recursive types in OCaml, use `type t1 = ... and t2 = ....`
   - This problem does not require much programming. Add a few lines and change a few lines.

2. You will implement three semantics for the silly Logo language described in `logo.pdf` by completing the code skeletons in `logo.ml`. For what it's worth, the file containing the sample solution is about 135 lines.

   (a) (Warmup)

   i. Complete the OCaml type `move` such that `move list` is a good representation of a Logo program.

   ii. Complete the OCaml function `makePoly : int->float->move`. It takes a *side-count* and a *side-length* and returns a `move` that makes the regular polygon with *side-count* sides and *side-length* side lengths. More precisely, the move (which should use a loop) does *side-count* forward and turn operations such that it visits each vertex of a regular polygon that starts at the current state and has one side "straight ahead from the current direction". The final state is the same as the start state (except for rounding errors).

   iii. Complete the OCaml function `scale : float -> move list -> move list`. It takes a scaling factor and a Logo program and returns a Logo program that is just like its input except all forward moves anywhere in the program have lengths multiplied by the scaling factor.

(b) Complete the OCaml function `interpLarge : move list -> (float * float) list`, a large-step interpreter for Logo. It returns the list of places visited in the order they were visited. A home or forward operation *always* adds to this list; a turn operation *never* does.

Hints:

- Just complete the recursive helper function `loop`, which should produce the places-visited list *in reverse order*. `loop` takes the current move-list, the current state, and the reversed list of places already visited.
- A program starting with a for-loop with $i > 0$ can be evaluated by evaluating its body appended to the program where i is decremented.

(c) Complete the OCaml function: `interpSmall : move list -> (float * float) list`, a small-step interpreter for Logo, by completing its helper functions `loop` and

```
interpSmallStep : move list -> float -> float -> float ->
                  move list * float * float * float
```

`interpSmallStep` takes a move list and a current state (x then y then *dir*) and returns a new move list and new state by "taking one small step". It raises an exception if passed the empty move list. `loop` uses `interpSmallStep` to build the places-visited list *in reverse order* by checking if the returned state has a different x or y than the passed state (add to the places-visited list *if and only if* x or y changes).

Hints:

- Use the same "trick" for for-loops as in your large-step interpreter.
- `interpSmallStep` is not recursive.

(d) In an OCaml *comment*, explain a reason that your small-step and large-step interpreters are *not* equivalent, i.e., the same Logo program may produce slightly different traces with the two intrepreters. Include at least one example program in your explanation. **Hint:** Think of useless moves.

(e) Complete the OCaml function

```
interpTrans : move list -> float -> float -> float ->
              (float * float) list * float
 (* i.e., move list -> (float -> float -> float -> (float * float) list * float) *)
```

a translational semantics for Logo (like we did for IMP). The returned function takes a program state and returns a list of places visited *and* the d in the resulting state. The returned function may use OCaml functions, lists, and arithmetic, but *not* the `move` type.

Hints:

- The empty move-list becomes a function that ignores two of its three arguments.
- A move-list starting with a non-loop becomes a function that "does the first move", passes the "new state" to the function that is the translation of the list tail, and then returns a (possibly-longer) trace and the tail's computed direction. We do *not* recommend building the trace in reverse order.
- For move-lists starting with for-loops, you should use a recursive OCaml function that takes an integer i and returns a function of type `(float->float->float -> (float*float) list * float)`. If $i = 0$, it just uses the translation of the tail of the list, else it "composes" the translation of the loop-body with the recursive function applied to $i - 1$.
  This "composing" (which we recommend putting in a helper function) is a little unusual: We need a function that takes two functions `f1` and `f2` and returns a new function that is the same as "running `f1` and then `f2`". That means:
  i. Running `f2` using the direction produced by `f1` and the last position in `f1`'s trace
  ii. Appending the traces
  iii. Returning the direction that `f2` returns

3. **(Challenge Problems)**

   (a) In a new file, further extend the large-step IMP interpreter to support the statement `pop` *str*. The values stored to *str* are conceptually in a stack and `pop` removes the shallowest stack element (so subsequent variable accesses will see the next stack element). In an OCaml comment, describe any corner cases not well explained by this definition and how you resolve them.

   (b) Write `canonicalize : heap -> heap` such that if heaps h_1 and h_2 are indistinguishable via lookups, updates, restores, and pops, then the results of calling `canonicalize` on them are structurally equivalent (OCaml's `=` operator returns true) and indistinguishable from h_1 and h_2.

   (c) Hook any of your Logo implementations up to a graphics library so that you can draw traces produced by a program. Optionally, add a "set-color" primitive and/or a "set-thickness" primitive to the Logo language so that you can draw more interesting pictures.