

CSE P505, Spring 2006, Assignment 2

Due: Tuesday 25 April 2006, 5:00PM

Last updated: Apr 21. Problem 2(f) is pretty tricky. You may use the Caml standard library as you wish.

1. Change 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`).

Begin Hints:

- This problem does not require much programming.
- Change the type `heap` to be mutually recursive with a new datatype you define. (To define mutually recursive types in Caml, use `type t1 = ... and t2 = ...`)

End Hints

2. You will implement three semantics for the silly Logo language described in `logo.pdf` by completing the code skeletons in `logo.ml`.
 - (a) Complete the Caml type `move` such that `move list` is a good representation of a Logo program.
 - (b) Complete the Caml 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).
 - (c) Complete the Caml 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.

Begin 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.

End Hints

- (d) Complete the Caml 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.

Begin Hints:

- Use the same “trick” for for-loops as in your large-step interpreter.

End Hints

- (e) In a Caml *comment*, explain two (related) ways your small-step and large-step interpreters are *not* equivalent, i.e., ways Logo programs may produce different traces with the two interpreters.

Begin Hints:

- Think of useless moves.

End Hints

- (f) Complete the Caml 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 Caml functions, lists, and arithmetic, but *not* the move type.

Begin 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 Caml 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 unusual: Roughly, one appends the two traces using the first element of the first trace as the initial x and y for the second function, but you need a special case when the first trace is empty. In any case, the second function’s direction result is the composition’s direction result.

End Hints

3. (Extra Credit)

- (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 a Caml 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 (Caml’s `=` operator returns true) and indistinguishable from h_1 and h_2 .

Turn in:

- Email your source code to Ben as attachments.
- **If you are using Seminal, please include your backup files.**
- Put your code in files `imp.ml` and `logo.ml`.
- If you do the extra credit, put it in `imp2.ml`.