Structure of Programming Languages, Assignment 1
Due: Thursday, 10/12

(in class, or before 2pm in CIS624 drop box in Deschutes)

Ensure you understand the course policies for assignments.
`hw1code.tar`, available on the course website, contains several OCaml files you need.

1. Add the following definitions to `trees.ml`. Do not use mutation. Do not change the type definitions or functions provided to you. Do not assume trees are sorted (though this is largely irrelevant).

   (a) Define `fromList` of type `int list -> inttree` to make a sorted tree containing exactly the ints in the list without repeats. Use `insert` (provided).

   (b) Define three functions, `sum1`, `prod1`, and `avg1`, to compute the sum, product, and average of the ints in a tree. Each has type `inttree -> int`. The sum and product of an empty tree are 0 and 1, respectively. For average, the empty tree should cause a `DivisionByZero` exception to be raised. For `sum1` and `prod1`, do not use helper functions. For `avg1`, do not traverse the tree more than once. (Hint: Have a helper function do the traversal and return a pair.)

   (c) Define `map` of type `(int -> int) -> inttree -> inttree` to produce a tree with the same shape as its second argument with the int at each position the result of applying the first argument to the int at the same position in the second argument.

   (d) Define `negateAll` of type `inttree -> inttree` using `map`. It produces a tree of the same shape where each int is replaced with its negation.

   (e) Define `sum2`, `prod2`, and `avg2` to compute the sum, product, and average of a tree (see above), but using `fold` (provided). You should not need more than 1 line (possibly 2 for `avg2`). Use the same pair technique for average.

   (f) In a *short English paragraph*, explain how a client of the `iter` function (provided) would use it to process all the ints in a tree. In a second *short English paragraph*, explain how `iter` is implemented (e.g., "when" and "how" it traverses the tree).[1]

   (g) Define `sum3`, `prod3`, and `avg3` to compute the sum, product, and average of a tree (see above), but using `iter` (provided). For product, the code must "stop as soon as it sees a 0" (this is easier than when using `fold`). Hint: You should need about 5 lines for each function. For each, use a local helper function as a "loop" that takes the iterator and the answer-so-far.

   (h) **Challenge Problem:** When writing functions that cannot always produce answers (e.g., average on an empty tree), one chooses between raising an exception or returning an option type. In this problem, you write short higher-order wrapper functions to "make the other choice":

   - Write `optionToException` of type `('a -> 'b option) -> exn -> 'a -> 'b`. It should return a function that takes an argument and raises the `exn` if and only if the `'a -> 'b option` returns `None` for the same argument.
   - Write `exceptionToOption` of type `('a -> 'b) -> (exn -> bool) -> 'a -> 'b option`. It should return a function that takes an argument and returns `None` if and only if the `('a -> 'b)` (when passed the same argument) raises an exception that causes the `exn -> bool` to return true.
   - In English, explain why `optionToException` takes an `exn` but `exceptionToOption` takes an `exn -> bool`.

[1] You might also try implementing `iter` using mutation instead of higher-order functions. It is not very pleasant.

2. We will extend IMP with the ability to push and pop heaps (material through lecture 3):

   - A program state is now $L; H; s$ where H and s are as before and L is a list (i.e., a stack of heaps).
   - There are two new statement forms: pushheap and popheap x. The former adds the current heap H to the beginning of L. (Informally, it "copies" H "all at once".) The latter replaces the current heap H with the first element of L *except* x maps to $H(x)$ and replaces L with the tail of L. If L is the empty list, then popheap x has no effect. Both pushheap and popheap x "become skip" in one step.

   (a) Extend the provided interpreter to support this new language. Hints/requirements:
      - Do not use mutation.
      - Change only `interp_step`, `iter`, and `interp`. (The parser is already written. The code for heaps and expressions does not need changing.)
      - Have `interp_step` and `iter` take a third argument (the list of heaps) and have `interp_step` also return a "new" list.
      - You need to change each case of `interp_step` and add two more cases corresponding to the two new statement forms (pushheap and popheap x).

   (b) Give an IMP program using pushheap and popheap x that we can use for testing. Note the parser will let you put parentheses around any expression or statement and requires parentheses in certain positions.

   (c) Define a formal small-step operational semantics for this new language. Hints/requirements:
      - The inference rules should have the form $L; H; s \to L'; H'; s'$.
      - Use the syntax $L ::= [] \mid H::L$ for lists.
      - Because the form of rules has changed, include "all the rules" (i.e, rules for assignment, conditionals, etc.)
      - You need 3 rules total for pushheap and popheap x.
      - You do *not* need to write the semantics for expressions because they have not changed.

   (d) In a short English paragraph, explain why our language would be much less useful if popping a heap did not copy one value from the current heap.

3. We will extend IMP with the ability to push and pop variables:

   - A heap now maps a variable to a number and a list (i.e., a stack) of numbers. The initial heap maps each variable to 0 and the empty-list (though it's not implemented that way).
   - There are two new statement forms: pushvar x and popvar x. pushvar x changes H such that x maps to the same number and a list that is like the old list except the number is at the beginning. popvar x is the inverse of pushvar x, i.e., it changes H such that x maps to the number at the beginning of the list and that number is then removed from the beginning of the list. popvar x has no effect if the list for x is empty. Both pushvar x and popvar x "become skip" in one step. Example: Given a heap H that contains `{(x1,1,[6;3]),(y1,-3,[2])}`, applying pushvar $x1$ will return a heap H' that contains `{x1,1,[1;6;3],(y1,-3,[2])}` and popvar $x1$ on H' will return a heap that contains `{x1,1,[6;3],(y1,-3,[2])}`.

   (a) Extend the provided interpreter to support this new language. Hints/requirements:
      - Do not use mutation.
      - Change only `get_var`, `set_var`, and `interp_step`, but also add functions `push_var` and `pop_var`, which take a heap and a string and return a heap. For `interp_step` you only need to add new cases. For `push_var` and `pop_var`, you typically produce a new heap with a different entry for the string. However, you must also do the right thing if there is no entry for the string, which is equivalent to there being an entry like `(str,0,[])` "before the change".

- Just "add on" to your solution the previous problem. This addition is completely orthogonal.

(b) Give an IMP program using pushvar x and popvar x that we can use for testing.

(c) **Challenge Problem:** Define a formal small-step operational semantics for this new language. Be sure to give a precise definition for heaps, explaining your notation.

4. We will prove a property of execution for IMP with pushheap and popheap x.

   (a) Formally prove the following: (For all s, L, H, and s',) If s has no while loops and $[]; \cdot; s \rightarrow^* L; H; s'$ (i.e., we run s for any number of steps), then the length of L does not exceed the number of occurrences of pushheap in s (i.e., in the original program). Hints:

      - Use induction on the number of steps taken, and a second inductive proof on the height of the derivation of taking one step.
      - For both proofs, you need to strengthen the induction hypothesis. Think through how to do so before writing out a long incorrect proof.

   (b) Formally prove the following: The previous claim is false if we allow s to contain while loops. Hint: Do not use induction.

5. **Challenge Problem:** Formally state and prove the following for our original IMP language: For any s that does not contain the variable `x`, the IMP programs `x:=x+1; y:=x*x;` s`; x:=x-1` and `y:=(x+1)*(x+1);` s are equivalent.

What to turn in:

- OCaml source code for problem 1 in a file called `trees.ml`.
- Hard-copy (written or typed) answers to problems 1f, 2c, 2d, and 4.
- OCaml source code for problems 2a and 3a in a file called `interp.ml`.
- IMP code for problem 2b and 3b in files called `2b.imp` and `3b.imp`.
- For the challenge problems, edit `trees.ml` and your hardcopy solutions.

Do not modify interpreter files other than `interp.ml`.