CSE 505, Fall 2003, Assignment 1
Due: 14 October, 10:30AM (firm)

Advice: This homework is large. Do problem 0 as soon as possible.

Code provided: Problems 1–4 involve extending and modifying an interpreter available on the course website. You do not need to understand the lexer (`lex.mll`) or parser (`parse.mly`), which already support the necessary extensions. The program takes one command-line argument, a file name holding an IMP program. To write IMP programs, obey these concrete-syntax rules:

- Nested statements (e.g., s in `while` e `(`s`)`) *must* be surrounded by round parentheses. This rule does not apply to s_1 nor s_2 in $s_1; s_2$.
- Parentheses around other statements and expressions is optional.

A trivial Makefile is included. It rebuilds completely each time and creates an executable called `interp`. (Yes, real O'Caml projects use real Makefiles, but rebuilding is easier to understand and takes about 1 second.)

0. (O'Caml Warm-Up)
   (a) Implement a binary search tree where each node holds an integer.
      i. Write an insert function that takes a tree and an integer and returns a new tree that includes the integer.
      ii. Write a lookup function that takes a tree and an integer and returns true iff the integer is in the tree.
      iii. Write a function that sums the integers in a tree.
      iv. Write a function that takes a tree t and an integer i and returns the least integer in the tree strictly greater than i (or raises an exception if i is greater than every integer in the tree).

   (b) Implement a binary search tree that is polymorphic; the nodes of the tree can hold data of any type α (but all the nodes of a tree hold the same type of data). The tree should also have a comparison function that takes two inputs of type α and indicates if the first is less-than, equal, or greater-than the other. Implement the same functions for this type as you did for integer trees, except there is no sensible way to sum the elements.

1. (Interpreter Warm-Up)
   (a) Describe in fewer than 200 English words how the code provided to you implements the heap used to interpret programs.
   (b) Replace the implementation of the heap with one where the empty heap is the O'Caml empty list (`[]`).
   (c) Describe the semantic difference between the interpreter provided and one where line 25 (commented `(*THIS LINE*)`) is `(h,Seq(s3,s2))`. Give an example IMP program that exhibits the difference.

2. (ntimes) In this problem, we extend IMP as defined in class with the statement form ntimes e (s). Informally, we evaluate ntimes e (s) by evaluating e to a number and then executing s that number of times. (If e evaluates to a nonpositive number, do nothing.)
   (a) Extend our operational semantics to include ntimes e (s), without introducing any new variables.
   (b) Extend our operational semantics to include ntimes e (s) by using only one rule and a new variable. (By "new variable," we mean you can assume you know an x that does not appear anywhere else in the program or the heap. Write the premise "x fresh" to indicate this assumption.)

(c) Prove or disprove the following:

i. For all H, s and e, if $H; s$ terminates, then $H;$ ntimes e (s) terminates.

ii. For all H, s and e, if ntimes e (s) terminates, then $H; s$ terminates.

iii. For all s and e, if for all H we know $H; s$ terminates, then $H;$ ntimes e (s) terminates.

iv. For all s and e, if for all H we know $H;$ ntimes e (s) terminates, then $H; s$ terminates.

(d) Extend the interpreter to implement ntimes e (s).

3. (Code Pointers) In this problem, we extend IMP as defined in class with code pointers. We add the statement forms $x := (s)$ and run x. Informally, the former stores the statement s in x and the latter runs whatever statement is currently in x. If x holds a number, then run x does nothing. If x holds a statement, then the expression x evaluates to 0.

(a) Extend our operational semantics to include $x := (s)$ and run x. (Hint: Extend the definition of heaps and $H(x)$. Replace the rule for variable lookup with two rules.)

(b) Extend the interpreter to implement $x := (s)$ and run x.

(c) Give an IMP program using $x := (s)$ and run x that we can use for testing.

4. (Approaches to Error Handling) In this problem, we no longer allow run x if x holds a *nonzero* number or the expression x if x holds a statement. Instead, these situations are now *errors*.

(a) Change the interpreter so that a terminating program either "exits normally" (and gives the value in `ans`) or "exits with an error" (by printing "run-time error!").

(b) Write an O'Caml function `prevent_error` that takes an IMP program and returns false if the program contains either of the following:

- statements of the form $x := e$ and run x for the same x.
- an expression x and and a statement of the form $x := (s)$ for the same x

(c) `prevent_error` is a static analysis that prevents errors, but is conservative. Give an example IMP program that does not produce an error even though `prevent_error` returns false.

(d) (For this problem only, you do not need to justify your answers.)

i. Yes or no: For all s, if `prevent_error` s evalutes to true, then are the interpreters you wrote for 3(b) and 4(a) are equivalent with respect to s?

ii. Yes or no: Can we write a more advanced version of `prevent_error` that is exact (returns false exactly when a program would produce a run-time error) and always terminates?

What to turn in:

- O'Caml source code for problem 0 in a file called `trees.ml`.
- Hard-copy (written or typed) answers to problems (1a), (1c), (2a), (2b), (2c), (3a), (3c), (4c), and (4d).
- O'Caml source code for problems (1b), (2d), and (3b) in a file called `interp1.ml`.
- O'Caml source code for problem (4a) and (4b) in a file called `interp2.ml`.

Email your source code to Andy.
Do not modify interpreter files other than `interp.ml`.