CSE 505, Fall 2005, Assignment 3
Due: Monday 14 November 2005, 5:00PM

`hw3.tar`, available on the course website, contains several Caml files you will need.
Last updated: October 31

Advice: Problems 1, 3c, and maybe 2a–c are excellent study problems for the midterm. (3c is "easy" for an extra credit.) 2d–f are not good review for the midterm. If you are going to skip a problem, skip 2e.

1. One definition of lists is in terms of "empty," "cons," "is-empty," "head," and "tail". This problem considers an alternative definition in terms of "empty,", "cons," and "fold". In Caml, "empty" is `[]`, "cons" is `::` (and is written infix), and "fold" is `List.fold_left`. More formally, consider the simply-typed λ-calculus with these syntactic extensions:

$$\begin{array}{rcl} e & ::= & \ldots \mid \mathsf{empty} \mid \mathsf{cons}(e,e) \mid \mathsf{fold}(e,e,e) \\ v & ::= & \ldots \mid \mathsf{empty} \mid \mathsf{cons}(v,v) \\ \tau & ::= & \ldots \mid \tau\ \mathsf{list} \end{array}$$

   (a) Extend our small-step operational semantics appropriately. Informally:
   - cons and fold should have call-by-value left-to-right behavior.
   - fold works like Caml's `List.fold_left`, which the Caml manual explains like this:
     `List.fold_left f a [b1; ...; bn] is f (... (f (f a b1) b2) ...) bn`
     Note that fold assumes the first argument evaluates to a (curried) two-argument function and the third argument evaluates to a list.

   Hints:
   - You need 5 "boring inductive rules".
   - You need 2 "interesting rules", one of which should (somewhat cleverly) use fold in the expression it produces.
   - If e_3 evaluates to the empty-list, then $\mathsf{fold}(e_1, e_2, e_3)$ should eventually evaluate to the result of evaluating e_2.

   (b) Describe all the new "stuck states" that our extensions introduce.

   (c) Give typing rules appropriate for our extensions. As in ML, the empty-list can have any list type and a list must have all its elements have the same type. Unlike ML, we do not have polymorphism, which is why empty, cons, and fold must be "built-into" our language, rather than defined in a library.

   (d) Prove the cases of the Preservation, Progress, and Substitution Lemmas relevant to our extensions.

   (e) Now forget about our formal stuff and just program with `[]`, `::`, and `List.fold_left` in Caml:
      i. Write `length` (to compute the length of a list) using only `List.fold_left` and addition.
      ii. Write `reverse` (to reverse a list) using only `[]`, `::`, and `List.fold_left`.
      iii. Write `map` (which has the same behavior as `List.map`) using only `[]`, `::`, `List.fold_left`, and `reverse`.
      iv. Write `is_empty` (to return true if and only if its argument is empty) using only `List.fold_left` and booleans.
      v. Write `head` (which returns `Some v` when passed a non-empty list starting with `v` and `None` when passed an empty list) using only `List.fold_left` and operations on options.
      vi. Write `tail` (which returns `Some t` when passed a non-empty list with tail `t` and `None` when passed an empty list) using only `List.fold_left`, `[]`, `::`, `reverse`, and operations on options.

   Note: Only `tail` took your instructor more than 80 characters. `tail` is a bit tricky.
   Note: In λ-calculus, `let f x = e x` and `let f = e` are equivalent if `e` terminates and has no side-effects. But in ML, if you want `f` to have a polymorphic type (as all the functions for this problem do), you cannot write `let f = e` when `e` is a function application. The reason has to do with mutation; we'll cover it in a few weeks.

2. (Implementation) The code provided to you defines abstract syntax and a parser for the simply-typed λ-calculus with booleans, pairs, and rec (instead of fix). To make parsing and type-checking easier, functions have the concrete form `(fn x:t.  e)`. Specfically, they must be surrounded by parentheses, they must have explicit argument types, and the ":" and "." must be present.

   Similarly, recursive functions have the form `(rec f:t x.  e)`. This creates a recursive function of type `t`. In `e`, `f` is bound to the function and `x` is bound to the argument.

   Conditionals and pairs require parentheses. For getting pair fields, use $(e).1$ and $(e).2$.

   (a) The file `adder` contains an incomplete test file. Complete the file so that running this program produces `((true,true),true)`. Hints:

   - We are treating pairs of the form $((b1, b2), b3)$ as 3-bit binary numbers where true is 1. After you provide the missing function, the provided code adds the two 3-bit numbers at the bottom of the file.
   - The function you add takes 3 bits and returns 2 bits. The first bit is the "result bit" of adding 3 bits (mod 2) and the second bit is the "carry bit" of adding 3 bits.
   - Sample solution is 4 rather dense lines, but an arguably simpler solution is about 15 lines.

   (b) Implement the type-checker for this language by changing `Main.typecheck`. Your type-checker should raise an exception if the expression does not type-check (or if it encounters the Closure variant, which is explained on the next page).

   (c) Implement a *large-step environment-based* interpreter by changing `Main.interpret`. Such an interpreter is explained on the next page. Your interpreter should raise an exception if it gets stuck. (But expressions that type-check should not get stuck). You will see that the code for implementing environments is provided to you. For this problem assume the value `PartC` is the first argument to `interpret`.

   (d) Change your interpreter so that when it is passed `PartD` it uses separate Caml threads (read about threads in the on-line manual) to evaluate the two subexpressions of a pair. Specifically, to evaluate (e_1, e_2):

   - Create an `exp option ref` that a spawned thread can mutate to store the result of its computation. Note this reference must be local because nexted pair-expressions will lead to multiple thread-creations.
   - Create a thread to evaluate e_2.
   - In the parent thread, evaluate e_1, then enter a loop (i.e, a recursive function) that checks whether the other thread has stored its answer into the `exp option ref` yet.[1] This is called "busy-waiting".

   (e) Change your interpreter so that when it is passed `PartE` it uses threads as in part (d), but it uses the `Condition` and `Mutex` libraries to avoid busy-waiting. Specifically:

   - The parent and child threads should shair a mutex and a condition variable.
   - The child should acquire the lock, signal the condition variable, and release the lock.
   - The parent should acquire the lock before entering a "waiting loop" and wait on the condition variable in the loop.

   (f) Change your interpreter so that when it is passed `partF` it uses threads as in part (d), but instead of looping, the parent thread uses `Thread.join` *after* evaluating e_1 to wait for the child thread to complete its work.

[1]On Cygwin but not Linux, your instructor found it necessary to put a call to `Thread.yield()` in the loop.

An environment-based interpreter does not use substitution. Instead, the program state includes an environment, which maps variables to values. To implement lexical scope correctly, functions are not values—they evaluate to *closures* (written $<e, E>$ below). Here is a formal large-step semantics using this approach (omitting booleans, conditionals, pairs, and rec). Pay particular attention to the rule for function application—we evaluate function bodies using the environment in its closure!

$$\begin{array}{rcl} e & ::= & c \mid x \mid \lambda x.\ e \mid e\ e \mid \mathsf{rec}\ f\ x.e \mid <\lambda x.\ e, E> \\ E & ::= & \cdot \mid E, x \mapsto v \\ v & ::= & c \mid <\lambda x.\ e, E> \end{array}$$

$$\frac{}{E; v \Downarrow v} \qquad \frac{}{E; x \Downarrow E(x)} \qquad \frac{}{E; \lambda x.\ e \Downarrow <\lambda x.\ e, E>}$$

$$\frac{E; e_1 \Downarrow <\lambda x.\ e_3, E_1> \qquad E; e_2 \Downarrow v_2 \qquad E_1, x \mapsto v_2; e_3 \Downarrow v}{E; e_1\ e_2 \Downarrow v}$$

As usual with large-step semantics, evaluation order is not fully specified. (We're CBV, but your implementation can be left-to-right or right-to-left.)

Formalizing rec without substitution is interesting. What we want is $E; \mathsf{rec}\ f\ x.e \Downarrow <\lambda x.\ e, E'>$ where $E' = E, f \mapsto <\lambda x.\ e, E'>$, which has a circularity. In math, we would take a fix-point over an environment-generator. In O'Caml, we need to either represent our environment with a recursive function or with a circular data structure (which requires mutation). The code given to you shows both approaches. (Note the first approach just pushes the problem off to O'Caml's implementors, making them be the ones who build a circular data structure.)

You can probably do the homework without understanding most of the previous paragraph.

3. **Extra Credit**

   (a) Read the proof in the textbook that all simply-typed λ-calculus programs terminate. Extend the proof for the language we defined in problem 1 (i.e, with empty, cons, and fold).

   (b) Extend the interpreter from problem 2 to support the "choice" operator $e_1 \parallel e_2$ (which the parser and abstract syntax support if you uncomment the obvious lines), which has this behavior:

   - If e_1 and e_2 terminate, then $e_1 \parallel e_2$ can behave like e_1 or like e_2.
   - If e_1 terminates and e_2 does not, then $e_1 \parallel e_2$ must behave like e_1.
   - If e_2 terminates and e_1 does not, then $e_1 \parallel e_2$ must behave like e_2.
   - If neither e_1 nor e_2 terminates, then $e_1 \parallel e_2$ must not terminate.

   Your solution should be elegant and use "very little if any" mutation. (If you use threads, you can use mutation for thread communication. If you don't use threads, don't use mutation.) Notice that there may be an arbitrary number of "nested" choice operators and they may not be "directly nested".

   Your instructor didn't do this problem but can imagine two approaches (you can do both for double extra-credit):

   i. Spawn threads for e_1 and e_2 and when one terminates, kill the other *and* any threads it may have spawned.

   ii. Use one thread, but after an expression inside a choice expands a "fix" expression, conceptually "stop" and switch to evaluating the other half of the choice. You can do this by rewriting $e_1 \parallel e_2$ as $e_2 \parallel e_1'$, where e_1' is just e_1 unless it contains nested uses of $\parallel$.

   (c) We extend the simply-typed λ-calculus-with-pairs with "fix2", which allows for two mutually recursive functions:

$$\begin{array}{rcl} e & ::= & \ldots \mid \mathsf{fix2}(e, e) \\ v & ::= & \text{(no change)} \\ \tau & ::= & \text{(no change)} \end{array}$$

   i. Extend our small-step operational semantics appropriately. Informally:
      - fix2 should have call-by-value left-to-right behavior.
      - For $\mathsf{fix2}(\lambda pair_1.\ e_1, \lambda pair_2.\ e_2)$, we create a pair of expressions out of the function bodies, with "the entire fix2" in place of $pair_1$ and $pair_2$. So to make a recursive call in e_1 or e_2, we get a field from the pair we are passed.

   ii. Give a typing rule for fix2. Hint: It's a bit messy, requring use of a τ_1, τ_2, τ_3, and τ_4.

   iii. Show that any program you can write with fix you can write with fix2. To be precise, give a semantic rule that transforms an expression using fix into one using fix2. Hint: You will need to use substitution.

   iv. Show that any program you can write with fix2 you can write with fix. To be precise, give a semantic rule that transforms an expression using fix2 into one using fix. Hint: You can use implicit renaming or substitution. The simplicity of the solution suggests that fix2 as defined here is a fairly dumb thing to add to a language.

What to turn in:

- Caml source code for 1e in a file called `firstname-lastname--hw3.ml`.
- Caml source code for problem 2 in a file called `firstname-lastname--main.ml`.
- Hard-copy (written or typed) answers to all other problems.
- If you do the Caml extra credit, do it in another file appropriately named.

Email your source code to Erika. Hard copy solutions should be put in Erika's grad student mailbox, in the envelope outside her office, or given to her directly.