CSE 505, Fall 2005, Assignment 1
Due: Friday 14 October 2005, 5:00PM

Ensure you understand the course policies on academic integrity (see the syllabus) and extra credit. `hw1.tar`, available on the course website, contains several Caml files you will need. Last updated: October 5

1. (Caml warm-up) Add the following definitions to `queues.ml`. Except for `enq3`, `deq3`, and `map3` (see below), you may not use mutation. Do not change the types definitions.

   (a) Define `mylist_rev` of type `'a mylist1 -> 'a mylist1` where the result is like the argument except the "list elements" are in reverse order.

   (b) Define `mylist_map` of type `('a -> 'b) -> 'a mylist1 -> 'b mylist1` where the "i^{th} element" of the result is the first argument applied to the "i^{th} element" of the second argument.

   (c) The type `'a queue1` can represent functional (first-in-first-out) queues:

   - The queue is empty when both lists are empty.
   - The front (soonest to be dequeued) of the queue is the first list, with the first element the front, the second element next-to-front, etc.
   - The back (last to be dequeued) of the queue is the second list, with the first element the back, the second element next-to-back, etc.

   i. Define `empty_queue1` of type `'a queue1` to be the value representing an empty queue.

   ii. Define `enq1` of type `'a queue1 -> 'a -> 'a queue1` (or a more general type) where the result is like the first argument except the second argument is the new back element.

   iii. Define `deq1` of type `'a queue1 -> 'a * 'a queue1` where the first component of the result is the front element of the argument and the second component of the result is like the argument except the front element is removed. Raise `DequeueEmpty` if the argument has no elements. Hint: Use `mylist_rev`.

   iv. Define `map1` of type `('a -> 'b) -> 'a queue1 -> 'b queue1` where the "i^{th} element" of the result is the first argument applied to the "i^{th} element" of the second argument.

   Note: If queues are "not reused" (i.e., no queue value is passed to `deq1` more than once), then every enqueue and dequeue operation has *amortized* $O(1)$ running time.

   (d) This problem is like the last one except we use `'a list` from the standard library in place of `'a mylist1`. See above for detailed descriptions. Hint: Use functions defined in the `List` library.

   i. Define `empty_queue2` of type `'a queue2`.

   ii. Define `enq2` of type `'a queue2 -> 'a -> 'a queue2` (or a more general type).

   iii. Define `deq2` of type `'a queue2 -> 'a * 'a queue2`.

   iv. Define `map2` of type `('a -> 'b) -> 'a queue2 -> 'b queue2`.

   (e) The type `'a queue3` can represent *imperative* queues:

   - A `ref` holds the queue's current contents, `None` if there are no elements, else a `Some` holding an `'a queue_impl` and a "pointer" to the "back" of the "same" `'a queue_impl`. The contents of the `'a queue_impl option ref` are always `None` (until `enq3` changes the contents).
   - An `'a queue_impl` represents a mutable list. The front of the list is the front of the queue.

   i. Define `empty_queue3` of type `unit -> 'a queue3` which returns a new imperative queue with no elements.

   ii. Define `enq3` of type `'a queue3 -> 'a -> unit` which changes its first argument to hold its second argument as its new back element.

   iii. Define `deq3` of type `'a queue3 -> 'a` which returns its argument's front element and changes its argument to remove this element.

   iv. Define `map3` of type `('a -> 'b) -> 'a queue3 -> 'b queue3` which creates a new imperative queue where the "i^{th} element" of the result is the first argument applied to the "i^{th} element" of the second argument. Raise `DequeueEmpty` if the argument has no elements.

2. (Interpreter Warm-Up)[1]

   (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 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.

3. (Non-preemptive Threads) We will extend IMP with threads:

   - A program state is now $H; s; q$ where H is the heap, s is the "currently running" thread, and q is a queue of waiting threads.
   - The grammar $q ::= \cdot \mid s{::}q \mid q{::}s$ describes queues, where $\cdot$ is the empty queue, $s{::}q$ has s as its front element, and $q{::}s$ has s as its back element.
   - The new statement form $\mathsf{spawn}(s)$ "becomes skip" after adding s to the back of the queue.
   - The new statement form yield "becomes skip" after putting the currently running thread at the back of the queue and removing the front queue element to make it the new currently running thread. (If the queue is empty, then yield just becomes skip.)
   - A program terminates when the state has the form $H; \mathsf{skip}; \cdot$.

   (a) Define a formal small-step operational semantics for IMP with threads. Hints/requirements:

   - The inference rules should have the form $H; s; q \rightarrow H'; s'; q'; b$, meaning $H; s; q$ becomes $H'; s'; q'$ and b is a *boolean* where true means the currently running thread "executed" a yield *and* the queue was non-empty (i.e., the currently running thread changed).
   - Most rules are like the rules for IMP except the queue is unchanged and the boolean is false.
   - You need 1 rule for when s is skip.
   - You need 1 rule for when s is $\mathsf{spawn}(s)$.
   - You need 2 rules for when s is yield.
   - You need 3 rules for when s is $s_1; s_2$. Two are like in IMP and one is for where s_1 changes the currently running thread (putting the old currently running thread at the back of the queue).

   (b) Extend the interpreter to support threads. Hints/requirements:

   - Do not use mutation.
   - Use your solution to problem 1(c) for the queue.
   - Change `iter` to take a queue argument.
   - Change `interp_step` to return a boolean for "did a yield" and a `stmt option` for a "just spawned" thread.
   - Do *not* change `interp_step` to take (or return) a queue argument. Instead, have `iter` "manage the queue".

   (c) Give an IMP program using $\mathsf{spawn}(s)$ and yield that we can use for testing.

4. (Language Properties) For the language you defined formally in the previous problem, prove or disprove each of the following claims.

   (a) If $H; s; q \rightarrow H'; s'; q'; b$, then q' and q have the same length.

   (b) If $H; s; q \rightarrow H'; s'; q'; b$, then q' and q have lengths that differ by at most one.

[1] This question was also used in the Fall 2003 offering of CSE505. Do not consult prior solutions to this problem, including the sample solution on the Web.

(c) If $H; s; q$, then either s is `skip` and q is $\cdot$ or there exist H', s', q', and b such that $H; s; q \rightarrow H'; s'; q'; b$. Hint: You will need a stronger induction hypothesis for when b is true.

5. (Static Analysis)

(a) Write a Caml function `spawn_count1` of type `stmt -> int` that returns how many `spawn` statements are syntactically in the argument.

(b) Write a Caml function `spawn_count2` of type `stmt -> int option` that returns `None` if a `spawn` statement appears in a while-loop body, else `Some` i where i is: 0 for skip, assignments, and yield; $1 + c$ for `spawn`(s) where c is the result for s; $c_1 + c_2$ for $s_1; s_2$ where c_1 and c_2 are the results for c_1 and c_2; and max(c_1,c_2) where c_1 and c_2 are the results for the two branches of an if-statement.

(c) True or false. If false, give an example program.

- If `spawn_count1` s is c, then during execution of s, the length of the queue is never more than c.
- If `spawn_count1` s is c, then during execution of s, the length of the queue must exceed c at least once.
- If `spawn_count2` s is `Some` c, then during execution of s, the length of the queue is never more than c.
- If `spawn_count2` s is `Some` c, then during execution of s, the length of the queue must exceed c at least once.

6. **(Extra Credit)**

(a) (Functional programming) The type `'a q` defines an "object-oriented" view of functional queues.

i. Define `qmaker1` of type `unit -> 'a q` as a "constructor" that creates a new empty queue, using `'a queue1` for its "hidden implementation".

ii. Define `qmaker2` of type `unit -> 'a q` as a "constructor" that creates a new empty queue, using `'a queue2` for its "hidden implementation".

iii. Define `qmaker3` of type `unit -> 'a q` as a "constructor" that creates a new empty queue, using `'a queue3` for its "hidden implementation".

iv. Define `is_functional` of type `(unit -> int q) -> bool`, which returns `true` if its argument constructs a functional queue and `false` if its argument constructs an imperative queue.

v. Consider `let f () = if Random.bool() then qmaker1 else qmaker2`. Explain whether or not it is possible for a caller to determine which function a call to `f` returns.

(b) (Threading and semantics)

i. Extend the semantics for IMP with threads to include nondeterministic pre-emptive scheduling, i.e., extend the operational semantics to capture that at any point any thread may become the currently-running thread.

ii. Extend the semantics for IMP with threads to include deterministic time-slice round-robin scheduling. In particular, change the operational semantics so that a thread that has not yielded for 42 steps is put on the back of the queue.

(c) (Threading and the interpreter)

i. Extend the interpreter for IMP with threads to include nondeterministic pre-emptive scheduling. In particular, after each computation step, stop the currently running thread with probability 0.01 and choose the next running thread uniformly at random from the available threads (including the thread just stopped). Do not use mutation (but do use the `Random` module, which is implemented with mutation).

ii. Extend the interpreter for IMP with threads to include deterministic time-slice round-robin schedule. In particular, change the interpreter so that a thread that has not yielded for 42 steps is put on the back of the queue. Do not use mutation.

What to turn in:

- Caml source code for problem 0 in a file called `queues.ml`.
- Hard-copy (written or typed) answers to problems 2a, 2c, 3a, and 4, and 5c.
- Caml source code for problems 2b, 3b, 5a, and 5b in a file called `interp.ml`.
- IMP code for problem 3c in a file called `example.imp`.
- Include extra credit (a) in `queues.ml` (except (v) is hard-copy). Extra credit (b) is hard-copy. For extra credit (c), turn in separate copies of the interpreter in files `interp_extra_i.ml` and `interp_extra_ii.ml`.

Email your source code to Erika as `firstname-lastname--hw1.tgz` or `firstname-lastname--hw1.zip`. The code should untar/unzip into a directory called `firstname-lastname--hw1`. Hard copy solutions should be put in Erika's grad student mailbox, in the envelope outside her office, or given to her directly.

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