CS 538

Project #2

Programming in Standard ML

Due: Friday, April 19, 2002

(Not accepted after Friday, April 26, 2002)

1. A widely-used data structure is the **priority queue**. A priority queue is an ordinary queue extended with an integer *priority*. When data values are added to a queue, the priority controls where the value is added. A value added with priority `p` is placed behind all entries with a priority ≤ `p` and in front of all entries with a priority > `p`. Note that if all entries in a priority queue are given the same priority, then a priority queue acts like an ordinary queue in that new entries are placed behind current entries.

   You are to write an SML abstract data type (an `abstype`) that implements a polymorphic priority queue, defined as `'a PriorityQ`. You may implement your priority queue using any reasonable SML data structure (a list of tuples might be a reasonable choice). The following values, functions, and exceptions should be implemented:

   - `exception emptyQueue`
     This exception is raised when `front` or `remove` is applied to an empty queue.
   - `nullQueue`
     This value represent the null priority queue, which contains no entries.
   - `enter(pri,val,pQueue)`
     This function adds an entry with value `val` and priority `pri` to `pQueue`. The updated priority queue is returned. As noted above, the entry is placed behind all entries with a priority ≤ `pri` and in front of all entries with a priority > `pri`.
   - `front(pQueue)`
     This function returns the front value in `pQueue`, which is the value with the lowest priority. If more than one entry has the lowest priority, the oldest entry is chosen. If `pQueue` is empty, the `emptyQueue` exception is raised.
   - `remove(pQueue)`
     This function removes the front value from `pQueue`, which is the value with the lowest priority. If more than one entry has the lowest priority, the oldest entry is removed. The updated priority queue is returned. If `pQueue` is empty, the `emptyQueue` exception is raised.
   - `contents(pQueue)`
     This function returns the contents of `pQueue` in list form. The front of `pQueue` is the leftmost element of the list, and the rear of `pQueue` is the last (rightmost) member of the list.
   - `size(pQueue)`
     This function returns the number of elements currently stored in `pQueue`.

2. (a) Assume we have a list `L` of integers. Define a function `listify(L)` that divides `L` into one or more sublists so that each sublist contains integers in non-decreasing (sorted) order. That is, if `v1` and `v2` are adjacent in `L` and `v1` ≤ `v2` then `v1` and `v2` are adjacent in the same sublist. However if `v1` > `v2` then `v2` ends one sublist and `v2` begins the next sublist. For example,

   `listify([3,5,1,8,9,2,1,0])` returns `[[3,5],[1,8,9],[2],[1],[0]]`
   `listify([1,2,3,4,5,6])` returns `[[1,2,3,4,5,6]]`
   `listify([5,4,3,2,1])` returns `[[5],[4],[3],[2],[1]]`

   (b) If the output of `listify` contains a single sublist, then we know the input list `L` was in sorted order. This makes it easy to test for duplicates within `L`—we just compare adjacent values. If listify returns more than one sublist, we can merge the first two lists into one sorted list, and repeat the process until we have one sorted list. Then testing for duplicates is easy. For example,

   `[[3,5],[1,8,9],[2],[1],[0]]` reduces to `[[1,3,5,8,9],[2],[1],[0]]`, then to `[[1,2,3,5,8,9],[1],[0]]`, then to `[[1,1,2,3,5,8,9],[0]]` and finally to `[0,1,1,2,3,5,8,9]`.

   Implement `duplicates(L)`, which sorts the values in `L` using `listify` as a subroutine and tests for duplicates.

   (c) Our algorithm for testing for duplicates is somewhat inefficient. If a duplicate appears in `L` it can readily be detected when `listify` initially partitions `L` or when the sublists produced by `listify` are merged.

   Create a function `duplicates2` (and whatever auxiliary functions you require) in which duplicate checking is *integral* to the implementation of `listify` and the merge component of the sort. As soon as a duplicate is seen, `true` should be immediately returned, without any further processing of `L`. If no duplicates are present, return `false` after the sort completes (but without any further checking). Use SML's exception mechanism to force a return when you see a duplicate value.

3. (a) Write an ML function that computes the following recursive function:

   f(0) = 1
   f(-1) = 0
   f(-2) = 0
   f(m) = f(m-3) + f(m-2) + f(m-1) for m ≥1

   (Be sure to remember that in ML ~ is unary minus and - is binary minus.)

   What are the values of f(28), f(29), and f(30)? How long does it take to compute each of these values?

   To start a "CPU timer" in ML use

```
val t =  Timer.startCPUTimer();
```

   To determine how much CPU time (in seconds) has elapsed since timer `t` was created use

```
Time.toReal(#usr(Timer.checkCPUTimer(t)));
```

   Estimate how long f(34) will take to compute (using your timings for f(28), f(29), and f(30)).

(b) In a functional language like ML without side-effects or assignments, the value of a function always depends solely on its arguments. This allows us to use the **memoizing** optimization. When a function is called, its arguments and result value can be recorded. If the function is called again with the same arguments, the stored result can be returned immediately.

Write an ML function `fastF(m)` that is a solution to part (a) using memoizing. What is the value of `fastF(34)`? How long does it take to compute?

4. (a) Recall that a **lazy list** is a useful structure for representing a long or even infinite list. In SML a lazy list can be defined as

```
datatype 'a lazyList =
   nullList | cons of 'a * (unit -> 'a lazyList)
```

This definition says that lazy lists are polymorphic, having a type of `'a`. A value of a lazy list is either `nullList` or a `cons` value consisting of the head of the list and a function of zero arguments that, when called, will return a lazy list representing the rest of the list.

Write the following SML functions that create and manipulate lazy lists:

- `seq(first,last)`
  This function takes two integers and returns an integer lazy list containing the sequence of values `first`, `first+1`, ... , `last`
- `infSeq(first)`
  This function takes an integer and returns an integer lazy list containing the infinite sequence of values `first`, `first+1`, ...
- `firstN(lazyListVal,n)`
  This function takes a `lazyList` and an integer and returns an ordinary SML list containing the first `n` values in the `lazyList`. If the `lazyList` contains fewer than `n` values, then all the values in the `lazyList` are returned.
- `Nth(lazyListVal,n)`
  This function takes a `lazyList` and an integer and returns an `option` representing the `n`-th value in the `lazyList` (counting from 1). If the `lazyList` contains fewer than `n` values, then `none` is returned. (Recall that `'a option` = `some of 'a | none`).

(b) Is is useful to remove unwanted values from a list using a **filter**. A filter, denoted as `filter(controlList,dataList)`, uses a boolean valued control list to select values from a data list; `true` signals that the corresponding data list value is to be kept; `false` signals that the corresponding data list value is to be deleted.
For example, `filter([true, false, false, true], [1,2,3,4])` = `[1,4]`. You are to program an SML version of `filter` that uses a lazy boolean list to filter a lazy data list; the result is a lazy list containing only data list values corresponding to true values in the control list.

(c) A wide variety of techniques have been devised to compute prime numbers (numbers evenly divisible only by themselves and one). One of the oldest techniques is the "Sieve of Eratosthenes." This technique is remarkably simple.

You start with the infinite list L = 2, 3, 4, 5, ... The head of this list (2) is a prime. If you filter out all values that are a multiple of 2, you get the list 3, 5, 7, 9, ... The head of this list (3) is a prime. Moreover, if you filter out all values that are a multiple of 3, you get the list 5, 7, 11, 13, 17, ... Repeating the process, you repeatedly take the head of the resulting list as the next prime, and then filter from this list all multiples of the head value.

You are to write a SML function `primes()` that computes a `lazyList` containing all prime numbers, starting at 2, using the "Sieve of Eratosthenes." To test your function, evaluate `firstN(primes(),10)`. You should get `[2,3,5,7,11,13, 17,19,23,29].` Try `Nth(primes(),20)`. You should get `some(71)`. (This computation may take a few seconds, and do several garbage collections, as there is a lot of recursion going on.)