Informatik and Mathematical Modelling 12/9 07
DTU Michael R. Hansen

02153 Declarative Programming
Programming Exercise 3

This exercise collection has parts:

1. A first part where the purpose is to make you more acquainted with recursion, basic types, lists and the use of libraries. This is a collection of small exercises.
2. The second part concerns efficient algorithms. In particular you shall develop two versions of merge sort, which both have a $n \log n$ worst case execution time.

Strive for succinctness and elegance when you solve the problems — it is important that your programs and program designs can be communicated to other people.

Part I

1. Use of Libraries. This exercise guides you through the use of libraries, such as `Math` and `String`. See also Appendix D in the textbook (HR), on the online documentation of the libraries from the homepage of MoscowML.

   To use names declared in a program library, e.g. `Math.pi`, you should *load* the library first. This is done as follows:

```
- load "Math";
> val it = () : unit

- Math.pi;
> val it = 3.14159265359 : real
```

   By *opening* the library `Math`, you can use `pi` and all other names declared directly:

```
- open Math;
> type real = real
  val cos = fn : real -> real
  .....
  val pi = 3.14159265359 : real
  .....
  val e = 2.71828182846 : real
  val sqrt = fn : real -> real

-  pi;
> val it = 3.14159265359 : real
```

2. Declare an SML function `pow: string * int -> string`, where:

$$\texttt{pow}(s, n) = \underbrace{s \,\hat{}\, s \,\hat{}\, \cdots \,\hat{}\, s}_{n}$$

3. Prime numbers

   (a) Declare the SML function

   `notDivisible: int * int -> bool`

   where $\texttt{notDivisible}(d, n)$ is true if and only if d is not a divisor of n. For example $\texttt{notDivisible}(2, 5) = \texttt{true}$, and $\texttt{notDivisible}(3, 9) = \texttt{false}$.

   (b) Declare the SML function `test: int * int * int ->bool`. The value of $\texttt{test}(a, b, c)$, for $a \leq b$, is the truth value of:

$$\begin{array}{ll} & \texttt{notDivisible}(a, c) \\ \text{and} & \texttt{notDivisible}(a+1, c) \\ \vdots & \\ \text{and} & \texttt{notDivisible}(b, c) \end{array}$$

   (c) Declare an SML function `prime: int -> bool`, where $\texttt{prime}(n) = \text{true}$, if and only if n is a prime number.

   (d) Declare an SML function `nextPrime: int -> int`, where $\texttt{nextPrime}(n)$ is the smallest prime number $> n$.

   (e) Declare an SML function `pr: int -> int list` such that `pr` n is the list of the first n prime numbers.

   (f) Declare an SML function `pr': int * int -> int list` so that $\texttt{pr'}(m, n)$ is the list of the prime numbers between m and n.

4. On slow sorting

   (a) Declare an SML function finding the smallest element in a non-empty integer list.

   (b) Declare an SML function `delete: int * int list -> int list`, where the value of $\texttt{delete}(a, xs)$ is the list obtained by deleting one occurrence of a in xs (when this is possible).

   (c) Declare an SML function which sorts an integer list so that the elements are placed in weakly ascending order.

Part 2: Merge Sort

Merge sort is an efficient algorithm for sorting a list of elements, which has a worst-case execution time of order $n \log n$.

A *merge* of two sorted lists, e.g. merge([1,4,9, 12], [2, 3 4, 5 10,13]) is a new sorted list, [1,2,3,4,4,5,9,10,12,13], made up from the elements of the arguments. This operation can be declared so that it has a worst-case running time proportional to the sum of the length of the argument lists. Declare such a function.

Top-down merge sort

The idea behind *top-down* merge sort is a recursive algorithm: take an arbitrary list with more than one element: $[a_1, \ldots, a_j, a_{j+1}, \ldots, a_n]$, split it around the middle position, say j, into two lists: $[a_1, \ldots, a_j]$ and $[a_{j+1}, \ldots, a_n]$. Sort these two lists and merge the results. The empty list and lists with one element are the base cases.

Declare a function for top-down merge sort in SML, which has a worst-case execution time of order $n \log n$. (Argue about the worst-case running time.) You may use the functions `take` and `drop` from the list library for the splitting of a list.

Bottom-up merge sort

The idea behind *bottom-up* merge sort is explained as follows:

1. Construct a list of one-element lists $[[a_1], \ldots, [a_j], [a_{j+1}], \ldots, [a_n]]$, from the original list $[a_1, \ldots, a_n]$.
2. Traverse the list repeatedly, where each traversal merge neighbouring pairs of lists. For example, after one traversal the list has the form:

$$[merge([a_1], [a_2]), merge([a_3], [a_4]), \ldots]$$

   This process will end with a list containing one sorted list.

Declare a function for bottom-up merge sort in SML, which has a worst-case execution time of order $n \log n$. (Argue about the worst-case running time.)

Which of the two merge sort programs would you prefer. (Provide argument.)