Functional Programming and Specification

Practical 1

This practical exercise is to be completed by 4pm on **Tuesday 8th February**. It will be marked but the mark will not contribute to the overall mark for the course. Please use Moscow ML to test your programs before you submit them, using the DICE `submit` command, like so:

> For UG3 students: `submit cs3 fps-3 1` *myfile*$_1$ *myfile*$_2$ … *myfile*$_n$
> For UG4/MSc students: `submit msc fps-4 1` *myfile*$_1$ *myfile*$_2$ … *myfile*$_n$

You may use any functions in the Moscow ML library.

Please use the exact type names and function names given below to allow some degree of automation to be used in testing your solution. (This is not the only way that your solution will be assessed, but it is a useful filter.)

Exercise 1 : BSTW compression

The BSTW algorithm, named after its inventors (Bentley, Sleator, Tarjan and Wei), compresses text. It is a one-pass algorithm that maintains a table containing all the words it has seen so far. When compression of a message begins, the table is empty. When a word that has not been seen before is encountered, it is added to the output and also entered at the beginning of the table. When a word that *has* been seen before is encountered, its *position* is added to the output instead of the word itself, and then its entry is moved to the beginning of the table, pushing all the other words down one place. Thus recently seen words will remain near the beginning of the table. This makes the algorithm "locally adaptive", taking advantage of the tendency for words to occur frequently for short periods of time then fall into long periods of disuse. The compressed message contains enough information for the uncompression algorithm to construct and maintain a copy of the compressor's table "on the fly".

Write ML functions implementing BSTW compression and uncompression. Your solution should include functions `bstw:string->bstw_message` and `unbstw:bstw_message->string` where

```
datatype element = Word of string | Index of int
type bstw_message = element list
```

You should assume that the original input contains no numbers. Don't worry about preserving the exact pattern of spaces and line breaks.

Exercise 2 : One-time passwords

One-time passwords are passwords that work once. After you've used such a password, you need to use a new password the next time. An *acceptor function* checks to make sure you are using the correct password; if so, it returns two items, namely:

1. your new password, for use next time; and
2. a new acceptor function for that password.

Otherwise it returns an error message.

Let's model one-time passwords using the following datatype:

```
datatype passaccept = Wrong | Good of int * (int -> passaccept)
```

We can think of one-time passwords as a sequence of values

`Good(`p_0`,`acc_0`)`, `Good(`p_1`,`acc_1`)`, …

where `Good(`p_i`,`acc_i`) : passaccept` for each i. The idea is that p_i is the password you use at the i^{th} step. If you use it correctly, then acc_i returns the next password along with the next acceptor function:

$$acc_i(n) = \begin{cases} \texttt{Good}(p_{i+1}, acc_{i+1}) & \text{if } n = p_i \\ \texttt{Wrong} & \text{otherwise} \end{cases}$$

We need to provide `Good(`p_0`,`acc_0`)` to start off the whole sequence.

1. Define a value `even : passaccept` that gives the password sequence of even integers starting from 0. **Hint:** Use an auxiliary function of the form

   ```
   fun check p i = if i=p then Good( ... , check ... ) else Wrong
   ```

2. Define a value `fib : passaccept` that gives the password sequence of Fibonacci numbers $1, 1, 2, 3, 5, 8, 13, \ldots$.

3. Define a value `rand : passaccept` that gives a random sequence of integer passwords between 0 and 1000, using functions from the `Random` library module.

⋆

If you want to learn more (no extra credit):

- Implement a version of BSTW that produces/consumes a sequence of bits or bytes — see `Word8` in the Moscow ML library. (In a real BSTW compression/uncompression program, a message is encoded as a sequence of bits where indices are distinguished from words and small indices take fewer bits than large indices.)

- There is a nice introduction to data compression on the web in `http://www.ics.uci.edu/~dan/pubs/DataCompression.html`. Try to implement some other compression methods in ML, for example Lempel-Ziv coding (the basis for Unix `compress` and `gzip`).

- You can read about S/KEY$^{\text{TM}}$, a widely known one-time password (OTP) scheme, at `http://homepages.inf.ed.ac.uk/dts/fps/papers/skey.ps`. It uses a "secure hash function" f that is easy to compute but infeasible to invert. The first password p_0 is $f^N(s)$, where s is a secret key and N is a number, and the i^{th} password p_i is $f^{N-i}(s)$. Thus $f(p_{i+1}) = p_i$. Since f cannot be inverted, an eavesdropper who discovers one password p_i cannot compute the next password p_{i+1}. One-time passwords are a solution to the problem of *authentication*, which is one aspect of computer security.