Programming Assignment A
Due on Saturday January 26, 2019 at 11:59 pm

The goal of this assignment is to implement LZ78 compression. This is similar to the popular LZW compression (it was a forerunner) but is a bit simpler in ML. Of course, we'd like the compression function to be polymorphic. That is, it can compress a list of any type of values (provided the values can be compared with equality). We will build this up in two steps.

Question 1. [20 points]
Implement a simple linear dictionary.

A basic necessity for LZ78 compression (and decompression) is a dictionary. That is, the ability to insert a key-data pair into a data structure and later look up the data based on the key. A hash table or balanced binary tree would be the best. For this assignment we will use a list (with linear time lookup). Next assignment we'll make this better.

Implement two functions `listdictfind` and `listdictadd`. They should be polymorphic. `listdictfind` should be of type `(''a * 'b) list -> ''a -> 'b option`. That is, its first (Curried) argument should be a list of key-data pairs (the dictionary). The second argument should be a key to be looked for (same type as the first type of the pairs). It should return either `NONE` indicating that the key is not in the list, or `SOME x` indicating that x is the data associated with the key. `listdictadd` should be of type `('a * 'b) list -> 'a * 'b -> ('a * 'b) list`. That is, the first argument should be a dictionary (list of key-data pairs) and the second argument the pair to be added. The function should return the new dictionary.

You do not need to implement the ability to remove things from a dictionary, just add and look-up.

Question 2. [60 points]
Implement LZ78 compression.

LZ78 compression works by reading in a sequence of values and outputting a sequence of index-value pairs. The algorithm keeps a codebook of value sequences, called codewords (even if the values are not characters). This codebook is a dictionary mapping sequences of values to indexes – starting at 1. As it is reading, LZ78 compares the current sequence to the codebook. As long the current sequence is in the codebook, it continues to add subsequent values to it. Once it adds a value that causes the current sequence not to be in the codebook, it outputs an index-value pair. In particular, the index is the index of the last sequence to be in the codebook and the value is the last value of this current sequence (the value that caused the sequence to fall out of the codebook). It then adds this new sequence to the codebook as a new codeword with the next largest index.

If even a single value cannot be found in the codebook, then an index of 0 is used to indicate that the value output is its own new indexed sequence. The algorithm then starts again with the next value in the input (but with a codebook of one more value).

As an example, if the input sequence is of characters, the compression builds a sequence of int-char pairs. Consider the string "aababbaba" (which of course would actually be a list of characters not a string, but that gets tedious to write). It would be compressed in the following steps:

1. The first character ('a') is read. It is not in the codebook (which is empty), so it is added to the codebook with index 1 and the algorithm outputs `(0,`a')`.
2. The second character ('a') is read. It is in the codebook, so the algorithm continues and reads the next character ('b'). The word "ab" is not in the codebook, so it outputs `(1,'b')` to indicate that the next sequence is the same as codeword 1, but with a 'b' tacked on the end. This new codeword "ab" is added to the codebook with index 2.
3. The next character ('a') is read. It is in the codebook, so the algorithm continues. The next character forms the word "ab" which is also in the codebook, so it continues to the next character ('b') which causes the target word to be "abb" which is not in the codebook, so the output `(2,`b')` is generated (meaning that the next characters are the same as codeword 2, plus the character 'b'). The codeword "abb" is added to the dictionary with index 3.
4. Finally, a similar sequence is repeated to output `(2,'a')` as the last output.

As a bit of a special case, if adding the very last value of the entire sequence produces a codeword, the algorithm is in a bit of a bind as it does not have an extra value to tack on the end. Therefore, instead, it outputs the index of the codeword for the sequence without the last value along with the last value. For example, if the string in the example above had been "aababbabb" then the last step would have built up the codeword "abb" which is in the dictionary. The final output could not be (3,'') as the second element in the pair is not defined. So, instead the algorithm should output (2,'b'). This special case only applies to the very last value in the sequence.

So for this assignment, you need to write the function `lz78e` (for LZ78 encode). It should be able to take any dictionary representation (in the next assignment, we will be replacing the dictionary with a different one, but you shouldn't need to change your `lz78e` function). To that end, it should have the type

```
('a, ('a -> 'b list -> int option), ('a -> 'b list * int -> 'a))
        -> 'b list
           -> (int * 'b) list
```

This means that it has a series of Curried arguments. The first tuple is

1. an empty dictionary (of type `'a`),
2. a look-up function (of type `'a -> 'b list -> int option`) that takes a dictionary (`'a`) and a sequence of values (`'b list`) and checks to see if it (the sequence of values) is in the dictionary (returning `NONE` if it is not and `SOME index` if it is), and
3. a function to add a value-data pair to the dictionary (it takes a dictionary – type `'a` – and a value-data pair – `'b list * int` – and returns a new dictionary of type `'a`). The other (Curried) argument is the sequence to be compressed (of type `'b list`). The output is a list of index-value pairs (type `(int * 'b) list`).

Basically, the first triple argument defines the dictionary type (an empty dictionary, a method to check the dictionary, and a method to add to the dictionary). The second argument is the list of values to be compressed. In the type definition above, `'a` refers to the type of the dictionary and `'b` refers to the type of the values to be compressed. The "keys" for the dictionary are of type `'b list` and the "data" for the dictionary are of type `int`.

To help you get started, the decompression algorithm has been supplied for you (also available in electronic form):

```
fun lz78d (emptybook,inbook,addbook) l =
let
  fun decode _ nil _ = nil
    | decode book ((ind,h)::t) c =
    let
      val str = if ind=0 then [] else
                case inbook book ind of
                     NONE => [] (* this should never happen! *)
                   | SOME s => s
    in
      foldl op::
          (decode (addbook book (c,h::str)) t (c+1))
          (h::str)
    end
in
   decode emptybook l 1
end;
```

It is somewhat simpler than the compression algorithm, but it has a similar type and also accepts the dictionary definition as the first three arguments. However, note it uses the dictionary in the opposite direction: the dictionary maps indexes to codewords[1]

[1]Technically, it maps indexes to reversed codewords which are then "unreversed" when they are added to the decoded strings. This is for code efficiency; you do not need to worry about it unless you want to understand how it works.

for decompression, whereas in compression it maps codewords to indexes. Your polymorphic dictionary functions make this possible without any additional coding.

Once you have defined this function, you can then define a compression algorithm that employs your list-based dictionary (from part a) by defining

```
fun lz78le l = lz78e ([],listdictfind ,listdictadd) l;
fun lz78ld l = lz78d ([],listdictfind ,listdictadd) l;
```

You should be able to check that the compress and decompression work. Here are some examples to try. You should also try your own examples.

```
- lz78le (explode "aababbaba");
val it = [(0,#"a"),(1,#"b"),(2,#"b"),(2,#"a")] : (int * char) list
- implode (lz78ld it);
val it = "aababbaba" : string
- lz78le (explode "aababbaba");
val it = [(0,#"a"),(1,#"b"),(2,#"b"),(2,#"a")] : (int * char) list
- implode (lz78ld it);
val it = "aababbaba" : string
- lz78le (explode "aababbabb");
val it = [(0,#"a"),(1,#"b"),(2,#"b"),(2,#"b")] : (int * char) list
- implode (lz78ld it);
val it = "aababbabb" : string
- lz78le (explode "hihihiyahiyahiya!");
val it = [(0,#"h"),(0,#"i"),(1,#"i"),(3,#"y"),(0,#"a"),(4,#"a"),(6,#"!")]
  : (int * char) list
- implode (lz78ld it);
val it = "hihihiyahiyahiya!" : string
- lz78le [1,5,5,5,5,1,5,5,5,5];
val it = [(0,1),(0,5),(2,5),(2,1),(3,5),(0,5)] : (int * int) list
- lz78ld it;
val it = [1,5,5,5,5,1,5,5,5,5] : int list
- lz78le (lz78le (explode "aababbabb"));
val it = [(0,(0,#"a")),(0,(1,#"b")),(0,(2,#"b")),(0,(2,#"b"))]
  : (int * (int * char)) list
- implode (lz78ld (lz78ld it));
val it = "aababbabb" : string
```

Submission Instructions:
Submit all functions that you wrote (`listdictadd`, `listdictfind`, and `lz78e`) as well as the supplied functions (`lz78d`, `lz78le`, and `lz78ld`) and any auxiliary functions you wrote in a **single `.sml` file**. *Your functions must have the names given in this assignment.*

Please comment your code and do not submit any other files (any additional information should be placed in comments in the .sml file).

Submit the single `.sml` file via gradescope.com.