This assignment continues our exploration of functional programming, recursion and SML. You are now ready to work in a richer context and apply your skills to larger and more complex problems.

Preparation and submission  As usual, copy the template and check files from the course Calendar page. Rename the template file to asgt03.sml, and put your solutions in it. When you have completed the assignment, run the check file. When all is well, upload your file through the course submission page.

1 Substitution Ciphers, Concluded

On the last two assignments, we worked with substitution ciphers based on cycles and pangrams. Another strategy for creating substitutions is to shuffle the letters randomly. It has the advantage of being less predictable than pangrams, but it makes the substitution harder to remember. Here is an example of a random translation.

\begin{verbatim}
  ABCDEFGHIJKLMNOPQRSTUVWXYZ
  PLUMNZFDEWQOVASTYRCI_JHKGXB
\end{verbatim}

With this translation, SNOW becomes CASH, ARISTOTLE becomes PRECISION, and SAGEHEN becomes CPFNDNA. Figure 1 shows an old child’s toy that combines an arbitrary permutation with a shift. Its encryption is not any better than a Caesar cipher because anyone can know the permutation. If you and your co-conspirator are to preserve secrecy, you must decide on a permutation and keep it secret.

http://dilbert.com/strip/
2017-01-28
After our work on previous assignments, encoding with random permutations is not hard. We have only to find a way to generate the random permutations. We will use a random number generator to do it. See the appendix to this assignment for instructions on how to create and use a random number generator.

1. [3 points] A good way to produce a random permutation of the elements in a list is with the Knuth shuffle. The idea is, given a list, to choose an element from it at random. That element will be the first element of the permutation. To get the rest of the permutation, take the chosen element out of the list and recursively shuffle the remaining elements.

You are to write a function knuthShuffle that has the following type signature. It takes a random number generator and a list and returns a random permutation of the list.

\[
\text{knuthShuffle : Random.rand -> 'a list -> 'a list}
\]

Observe that you will have to first create the random number generator and then pass it as a parameter to the knuthShuffle function.

The implementation in SML is a natural list recursion, but it is a little different from our usual pattern. Instead of taking the first element and the rest of the list, \(x\) and \(xs\), we take a random element and a list with the rest of the elements. The recursive call is still on a list that is shorter than the original list, so we have a correct recursive structure.

Here is a framework:

\[
\begin{align*}
\text{fun knuthShuffle gen lst =} \\
\quad & \text{if length lst < 2} \\
\quad & \quad \text{then lst} \\
\quad & \quad \text{else random-elt :: (knuthShuffle gen other-elts);} \\
\end{align*}
\]

To get the random-elt and the list of other-elts you will need to use the random number generator and some functions from the List package. Read through the documentation for the List package at http://sml-family.org/Basis/list.html. You can use the functions by typing the package's name and then the name of the function. For example, typing

\[
\text{List.nth (["A", "B", "C", "D"], 2)};
\]

into SML results in a response of

\[
\text{val it = "C" : char}
\]

Once you understand what the functions do and how to call them, implement the knuthShuffle function as described above.
Postmortem on substitution ciphers  You might think that using random substitutions makes the code system more secure. It does so in the sense that there are more possible keys, making a brute force attack of trying all possible keys more difficult. But the key is harder to remember, and you may be tempted to write down the substitution thereby increasing the possibility that the information will leak out. Further, you and your coconspirator must meet or find another way to agree on a key. Keeping one’s encryption keys secure is a big challenge for computer security professionals.

Any substitution cipher, including one with a random key, has other weaknesses as well. It is subject to a letter-frequency attack. The most frequently occurring letters in an encoded message probably map to the most commonly used letters in English, E or T. The least frequently occurring letters are likely to be Q or Z. Further, if the plaintext for one encoded message becomes known, then the substitution—or a significant part of it—is also known. For these reasons, substitution ciphers are seldom used in real applications. We will see an example of a more secure encryption scheme later in the course.

II  Mastermind

Mastermind is a board game with two players—a codemaker, who creates a secret code by choosing a sequence of four colored pegs; and a codebreaker, whose object is to discover the code in as few turns as possible. On each turn the codebreaker makes a guess, and the codemaker responds with the number of exact matches (in which a color appears in the same place as in the secret code) and the number of inexact matches (in which a color appears in the secret code, but not in the same place). For example, if the secret code is

[Red, Blue, Yellow, Yellow]

and the guess is

[Yellow, Red, Green, Yellow]

there is one exact match (for the Yellow at the end) and two inexact matches (for the Red and the other Yellow).

The goal of the first part of this assignment is to create some functions that will allow the computer to play the role of the codebreaker. The algorithm we suggest is admittedly uncreative, to say the least, but the exercises will give you practice in manipulating lists. We will return to Mastermind later in the course.

The template file contains a new datatype for pegs and a list of all the six colors.
datatypes peg = Red | Orange | Yellow | Green | Blue | Violet;

val allColors = [Red, Orange, Yellow, Green, Blue, Violet];

A secret code will be a list of elements of type peg. In the real board game, there are four pegs in a code. We will work more generally and not put a limit on the size of a code. It is convenient to have a function that will generate all possible codes of a given length, and such a function appears in the template file. It has this type signature:

fun allCodes : int -> peg list list

See an appendix to this assignment for the details on implementing allCodes.

2. [2 point] Write a function exactMatches that takes a secret code and a guess and returns the number of exact matches. (We read the lists from left to right. If one list is longer than the other, the trailing elements of the longer list are ignored.)

   exactMatches : ''a list -> ''a list -> int

3. [3 points] a. Write a function countColors that takes a code and returns a list with the number of pegs of each color. It is convenient to order the elements of the list according to the enumeration of colors in allColors. For example, countColors [Red,Blue,Yellow,Yellow] returns [1,0,2,0,1,0].

   countColors : peg list -> int list

b. Write a function totalMatches that takes a secret code and a guess and returns the number of matches—exact or inexact.

   totalMatches : peg list -> peg list -> int

4. [1 point] Use results from previous problems to write a function matches that takes the role of the codemaker. Given a secret code and a guess, the function returns an ordered pair of integers—first the number of exact matches and then the number of inexact matches.

   matches : peg list -> peg list -> int * int

Notice that the result when matches is applied to a secret code it has type

peg list -> int * int

That function behaves as the codemaker, taking a guess and returning an ordered pair of integers. We will call such a function a codemaker function and use it shortly.
5. [2 point] When a codebreaker makes a guess and receives a response, the codebreaker is able to eliminate several secret codes. For example, if the guess is all Red pegs and the response is (1, 0), then the codebreaker knows that the secret code has exactly one Red peg. Codes with no Red peg, or more than one Red peg, can be excluded from consideration.

We say that a code is consistent with a guess and a response if the function matches returns the given response for that code and guess. For example, suppose that the guess was [Blue, Red, Blue, Blue] and the codemaker returned (1,1). Then we have, among many other values,

- matches [Orange, Blue, Blue, Green] [Blue, Red, Blue, Blue] returns (1,1)
- matches [Blue, Green, Red, Green] [Blue, Red, Blue, Blue] returns (1,1)
- matches [Orange, Blue, Green, Green] [Blue, Red, Blue, Blue] returns (0,1)

The codes [Orange, Blue, Blue, Green] and [Blue, Green, Red, Green] are consistent with the guess [Blue, Red, Blue, Blue] and response (1,1), but [Orange, Blue, Green, Green] is not.

Write a function isConsistent that takes a guess, a response, and a candidate code and returns a boolean value telling whether the candidate is consistent with the guess and response.

\[
isConsistent : \text{peg list} \rightarrow \text{int} \times \text{int} \rightarrow \text{peg list} \rightarrow \text{bool}
\]

Notice that, because isConsistent is curried if we give it the first two arguments, i.e. a guess and the corresponding response, then we get a function of type

\[
\text{peg list} \rightarrow \text{bool}
\]

The function is a predicate that tells us whether a candidate answer code is consistent the current guess; it will be used shortly—in the next problem!—to filter lists of possible secret codes. Make sure that you understand this!

6. [2 point] The next step in our strategy is to “thin out” a list of potential candidates for the secret code. Write a function filterCodes that takes a guess, a codemaker’s response to that guess, and a prior list of candidates. It returns a list of those candidates that are consistent with the given guess and the response to it.

\[
\text{filterCodes : peg list} \rightarrow \text{int} \times \text{int} \rightarrow \\
\text{peg list list} \rightarrow \text{peg list list}
\]

Suggestions: Take some time to understand the type signature. This function will mostly be a combination of previous functions. You may use the built-in function List.filter.

7. [4 points] Here is our strategy—if you can call it that! The codebreaker keeps a list of the codes that, at that point, are known to be possible. At the beginning, the codebreaker knows nothing, and all
codes are possible. At each step, the codebreaker “guesses” the first
code in the list of possibilities. Based on the response, the codebreaker
either knows the guess is correct (how?) or can filter the list and go on
to make another guess.

a. Write a function codebreaker that plays the role of the code-
breaker. It takes three arguments:

- a codemaker function, as described in Problem 4;
- the list of previous guesses in reverse order; and
- the current list of candidate codes.

At each step, codebreaker will either declare success and return the
list of guesses ending with the secret code, or it will apply itself recur-
sively with a longer list of guesses and a shorter list of candidates.

codebreaker : (peg list -> int * int) -> peg list list ->
peg list list-> peg list list

Picky point: The actual solution will always be in the list of possi-
ble solutions, so the function codebreaker should never encounter
an empty list of candidates, but you will get a warning if you omit the
empty pattern from your function. Declare an exception InternalInconsistency
and raise it when codebreaker encounters an empty list of candidate
solutions.

b. Write a function solve that takes a secret code and returns the se-
quen ce of guesses that our so-called strategy would make to discover
it. Assume that the codebreaker knows the length of the secret code
and uses that value to generate the initial list of possible solutions.

solve : peg list -> peg list list

Postmortem on Mastermind  Our work with Mastermind was intended
to provide practice with lists and recursion. A separate question, one
that is natural to ask, is “How good is our simple strategy?” Many
research papers have been written on the subject. With six colors
and four pegs, the configuration of the commercial board game, the
maximum number of steps that our strategy will take to find the
solution is nine. The average number of steps, over all possible secret
codes, is 5.765. A more sophisticated algorithm, due to Donald Knuth,
guarantees a maximum of five steps and an average of 4.476. We may
have a look at Knuth’s algorithm later in the course. You may study
similar strategies for other games in an artificial intelligence or an
algorithms course.

Even though our strategy does not do as well as other computer
generated ones, it is apparently comparable to that of typical human
players. The board pictured in Figure 2 has space for twelve guesses,
several more than the nine required by our strategy. It would be an unusual human being who begins a game of Mastermind by making a list of all 1296 possible solutions.

### III Representing Numbers in Other Bases

We normally write numbers in decimal notation, base 10. There is nothing special about 10 except that it is widely used today. The Aztec, Mayan, and Celtic civilizations used the vigesimal system, base 20. The Sumerians and Babylonians used the sexagesimal system, base 60. In computer science, we sometimes use the binary system, base 2, and the hexadecimal system, base 16. The octal system, base 8, is occasionally used as well.

Let us consider systems of representing numbers using bases between 2 and 20, inclusive. For the digits beyond 9, we will use letters, starting at A and omitting I because it is too easily confused with the digit 1. The value of the base is often called the radix. We will specify the radix, when necessary, as a decimal subscript. For example, the following are different representations for the same number:

\[ 101111_2 \quad 57_{10} \quad 47_{12} \quad 3B_{16} \quad 2F_{16} \quad 27_{20} \]

For Problems 8 through 10 we will assume radix values between 2 and 20 and raise an exception for any of the functions that receive input that violates this assumption.

8. [1 point] Write a function called `digitToChar` that takes as input a number representing a single digit and returns the character that corresponds to that digit. For example,

\[
\begin{align*}
\text{digitToChar 8} & \text{ returns the character #"8"}, \\
\text{digitToChar 11} & \text{ returns the character #"B"}.
\end{align*}
\]

This function should only be called with digits for radix values up to 20, inclusive. Declare and raise an exception `RadixException` if you receive a digit that is not valid for that radix range.

\[
\text{digitToChar : int -> char}
\]

9. [1 point] Write the inverse function `charToDigit` that takes a character representing a digit and returns its numerical representation. For example, `charToDigit #"B"` returns 11. Raise the `RadixException` if the input character does not represent a valid digit given our radix constraint of at most 20, inclusive.

\[
\text{charToDigit : char -> int}
\]
charToDigit : char -> int

10. [2 points] Now that we have digit-level functions, we can use them to write functions that process numbers. Write a function fromRadixString that takes a radix and a string of “digits” and produces the corresponding integer. For example,

\[
\text{fromRadixString} (3, "1202") \text{ returns the integer 47, and} \\
\text{fromRadixString} (18, "3H") \text{ returns 71.}
\]

Raise RadixException if the radix is not between 2 and 20, inclusive; if the string is empty; or if any of the “digits” in the string are out of range for the given radix. Allow the SML system to raise the exception Overflow if the string represents an integer that is too large for the type int.

\[
\text{fromRadixString} : \text{int} * \text{string} \rightarrow \text{int}
\]

11. [2 points] Write the inverse function toRadixString that takes a radix and an integer and produces the string representation of the integer in the specified radix. For example,

\[
\text{toRadixString} (3, 47) \text{ returns the string "1202", and} \\
\text{toRadixString} (18, 71) \text{ returns "3H".}
\]

Raise the exception RadixException if the radix is not between 2 and 20, inclusive, or if the integer is negative. Non-zero values should be represented without leading zeroes. Zero should be represented by a string with a single zero. Write your function from scratch without using any built-in conversion functions.

\[
\text{toRadixString} : \text{int} * \text{int} \rightarrow \text{string}
\]

Postmortem on number representation In SML, the largest value for an int is \(2^{30} - 1\), or 1073741823. On current 64-bit processors, the largest integer is \(2^{63} - 1\), a considerably larger number that is nearly \(10^{19}\) but is still too small for many purposes. To get around the limitation, programmers use packages that represent very large integers as lists (or arrays) of “digits” with a very large radix. For example, we could take the radix to be 65,536, and our numbers would be represented by lists of “digits” that range from 0 through 65,535. In SML, there is a datatype IntInf that is implemented in that way, only with an even larger radix.

it is very late
harry saved me from myself
i am mastermind

Sam Rubin '19, CS52 poet laureate, acknowledging a mentor’s help
Appendix I: Random Number Generators

For Problem 1, you will need to generate random numbers. Informally speaking, a sequence of numbers is random if the next number is not predictable from its predecessors. True randomness is difficult to find. People who are serious about random numbers often use natural phenomena, like cosmic rays or quantum events, to generate sequences of bits. For many uses, however, pseudo-random numbers that are generated by an algorithm are adequate. Most modern programming languages have facilities for generating pseudo-random numbers using an algebraic formula that produces sequences that will eventually repeat, but have no apparent pattern in the short run. The generators start with a seed and generate numbers based on that value. If two people use the same seed, they will get identical sequences of numbers. To avoid such duplication, people often take a reading of the computer’s clock and use the low order bits that represent tiny fractions of a second for the seed.

In SML, we create a random number generator with two integers as seeds.

```sml
val generator = Random.rand(47,42);
```

Don’t use 47 and 42. Pick your own seeds.

The generator has type Random.rand. Using the generator, we can generate sequences of pseudo-random numbers.

```sml
Random.randInt : Random.rand -> int
Random.randNat : Random.rand -> int
Random.randReal : Random.rand -> real
Random.randRange : (int * int) -> Random.rand -> int
```

The function Random.randRange will be most useful for us here. The call

```sml
Random.randRange (0,6) generator
```

will produce a pseudo-random value between 0 and 6, inclusive.

Appendix II: Generating All Lists of a Given Length

In our work with Mastermind, we need a way to generate all the possible secret codes in a game. It is an instance of the more general problem of finding all $k$-element lists whose elements are taken from a given list. We will create a function possibilities to carry out the task.

```sml
possibilities : 'a list -> int -> 'a list list
```

For example,
• possibilities \([0,1,2]\) 2 returns
  \([0,0],[0,1],[0,2],
  [1,0],[1,1],[1,2],
  [2,0],[2,1],[2,2]\);
• possibilities \([0,1,2,3]\) 12 returns a list with \(4^{12}\), or nearly
  17 million, twelve-element lists; and
• possibilities allColors 4 returns a list of all the secret
codes in the four-peg version of Mastermind.

The idea in implementing possibilities is to use recursion on the
integer argument \(k\). We can distinguish three cases:

• If \(k\) is negative, then the result is the empty list \([]\).
• When \(k\) is zero, we are asking for all lists of zero length. There is
  exactly one such list, the empty list, and our result must contain
  that list. The resulting list of possibilities is \([[]]\).
• When \(k\) is positive, we will generate a recursive result for \(k-1\)
  and use it to construct the result for \(k\). We can use consAll
  from Assignment 1 to prepend a single value to every list in the
  recursive result. We can then map that function to obtain a list of
  such lists, one for each possible first element. And finally we can
  append them all together.

Study the function below and verify that it matches the description
just given.

\[
\begin{align*}
\text{fun possibilities elts } k &= \, \\
&\quad \text{if } k < 0 \, \\
&\quad \quad \text{then } [] \\
&\quad \text{else if } k = 0 \, \\
&\quad \quad \text{then } [[]] \\
&\quad \text{else List.concat (map (consAll(possibilities elts \(k-1\))) elts)}; \\
\end{align*}
\]

Here, we have used the built-in function List.concat which appends
all the elements of a list of lists. In class, you may have seen a function
that does the same thing; it may have been called appendAll. Now it
is easy to declare the function allCodes that appears in the template
file.

\[
\begin{align*}
\text{val allCodes } &= \text{possibilities allColors}; \\
\end{align*}
\]

References and Credits

The image in Figure 1 is from the Scottish Rite Masonic Museum & Library—http://nationalheritagemuseum.

The image in Figure 2 is licensed under CC BY-SA 2.0 via Commons—https://commons.wikimedia.org/wiki/File:
Mastermind.jpg#/media/File:Mastermind.jpg.