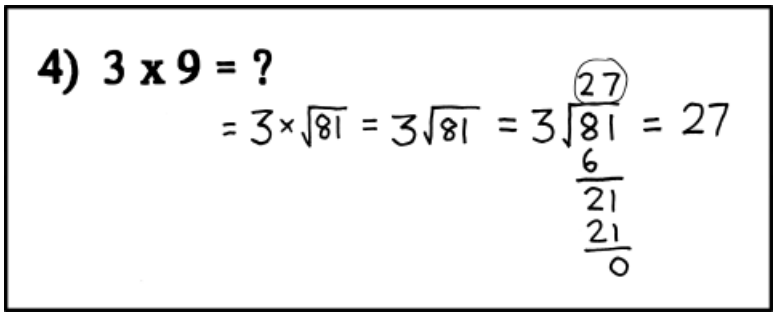
Computer Science 54

Assignment 2

Sunday, September 18 @ 11:59pm



https://xkcd.com/759/

In this assignment, we will be exploring writing more interesting functions in SML. For each function, make sure that you understand the type signature and what the function is doing. Try and work incrementally by playing with subparts of your function in the SML shell.

Partners On this assignment you may (and I'd encourage you to) work with a partner. If you choose to, you must both be there anytime you're working on the assignment.

Submission When you're all done, run the check file. If all is well, submit your assignment via Gradescope. If you worked with a partner, make sure that both names are at the top of the file. Only one person needs to submit, but make sure you add the other person when you submit.

1. **[1 points]** Warming up!

Write a curried function `count` that takes as in put a list and a value and counts the number of times that item occurs in the list

```
count: ''a list -> ''a -> int
```

Depending on how you write this, you will likely get the warning `Warning: calling polyEqual`. This is one of the few warning you can *ignore* in this class.

For example,

```
- count [0, 1, 0, 1, 0, 0] 0;
val it = 4 : int
- count (explode "banana") #"a";
val it = 3 : int
```

2. **[1 point]** Takeuchi

The Takeuchi function was developed in the 1970's as a benchmark for Lisp systems. It makes a large number of recursive calls without generating large integers. Write a function `tak` to compute the Takeuchi function.

$$\text{tak}(x, y, z) = \begin{cases} y & \text{if } x \le y, \text{ and} \\ \text{tak}(\text{tak}(x-1, y, z), & \\ \quad \text{tak}(y-1, z, x), & \\ \quad \text{tak}(z-1, x, y)) & \text{otherwise.} \end{cases}$$

```
tak : int * int * int -> int
```

3. **[2 points]** Zip it up!

The built-in function `ListPair.zip` takes a pair of lists and returns a list of pairs. If the lists are of unequal lengths, the trailing elements of the longer list are discarded. Write your own version of the function, called `myZip`.

This time, the recursion will be carried out on *both* arguments simultaneously. *Think about the base case(s).*

```
myZip : 'a list -> 'b list -> ('a * 'b) list
```

```
- myZip [1, 2, 3] [4,5];
val it = [(1,4),(2,5)] : (int * int) list
```

4. **[2 points]** What goes around comes around

Write an SML function `cycle` that rotates the elements in a list by a given amount. For example, `cycle 1` is a function that removes the first element of a list and places it at the end. The call `cycle 2` is equivalent to two consecutive calls to `cycle 1`.

If the argument `n` in `cycle n` is zero or another multiple of the length of the list, then `cycle n` returns the list unchanged. If the argument `n` is negative, then `cycle n lst` and `cycle (length lst + n) lst` give identical results.

Observe that the recursion is on the integer, not the list.

```
cycle: int -> 'a list -> 'a list
```

```
- cycle 3 [10, 20, 30, 40];
val it = [40,10,20,30] : int list
- implode (cycle 4 (explode "I love CS"));
val it = "ve CSI lo" : string
```

There are many ways to solve this problem, but one is to utilize an auxiliary function that simply cycles the list by one element. You can then use that function in your general `cycle` function. The approach has the added benefit that you can debug your auxiliary function independently.

5. **[2 points]** I think I cons, I think I cons, ...

Write a function `consAll` that takes a list of lists and an element, and prepends the element to every member of the list of lists.

```
consAll : 'a list list -> 'a -> 'a list list
```

```
- consAll [[1,2], [], [3]] 8;
val it = [[8,1,2],[8],[8,3]] : int list list
```

The order of the arguments to `consAll` may seem strange, but it will be convenient when we use `consAll` in a later assignment.

6. **[2 points]** Is it even or green?

The built-in function `List.filter` takes a predicate and a list and returns the sublist consisting of those elements that satisfy the predicate. For example, if one had a function `isGreen` that returned true whenever its argument was green, then

```
List.filter isGreen
```

would be a function that takes a list and returns a (possibly shorter, possibly even empty) list with only green elements.

A *predicate* is a function that returns a boolean value. An argument *satisfies* the predicate if the predicate returns `true` for that argument.

For another example.

```
List.filter isEven
```

is a function that "filters out" the non-even elements of a list.

Flashback to assignment 0!

```
- myFilter isEven [1, 2, 3, 4, 5, 6];
val it = [2,4,6] : int list
```

An important thing to remember is that the first argument to `List.filter` is a *function.*

Write your own version `myFilter` of the `List.filter` function.

```
myFilter : ('a -> bool) -> 'a list -> 'a list
```

The type signature for `myFilter` shows that it is a curried function with two arguments where the first argument is a *function*. Make sure you understand this.

Substitution Ciphers

For millennia, people have used encryption to send secret messages. We will encounter some of the techniques as the course progresses. In all cases, the *plaintext* is the human-readable message that is to be kept secret. The *ciphertext* is the encoded information that is, one hopes, inaccessible to anyone who does not know how to decrypt it. Often there is a number or a string or an algorithm, called the *key,* that allows someone to decode the cipher text.

The Caesar cipher, or shift cipher, is one of the simplest forms of encryption. We choose a constant shift distance d and replace each letter by its dth successor. The strings below show the substitutions for a shift of 4.

I come to encrypt Caesar, not to praise him.
The plaintext that men utter lives after them;
The ciphertext is oft interred with their bones;
—with apologies to W. Shakespeare

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ␣
EFGHIJKLMNOPQRSTUVWXYZ␣ABCD
```

The letter `A` becomes `E`, `B` becomes `F`, and so on. When we reach the end of the alphabet, we wrap around to the beginning. To keep things simple, we use only uppercase letters plus the blank space, denoted ␣. When we encrypt the blank space, we hide the word structure of the message and make it harder to decrypt. Many child's toys and newspaper puzzles do not encrypt blank spaces, making the messages easier to decrypt. Figure 1 shows a piece of contemporary jewelry that encrypts messages using a Caesar cipher.

Caesar ciphers are easy to decipher because all you need to know is the translation of one letter. That reveals the shift and all the letter translations are known. With our alphabet, including the blank space, there are 27 different shifts. One of them, the zero shift, does not change the message at all and is useless for secrecy. Thus there are 26 possible keys—much too small a number. An adversary could easily try all 26 possibilities and decrypt a message. We will study more sophisticated and secure ciphers in subsequent assignments.

Figure 1: A "computer" for a Caesar cipher. The inner ring of letters can rotate to produce any possible shift. We encrypt from the inner ring to the outer. The position shown is for a shift of 25; the letter `C` is translated to `B`. As of January 2017, the device is available for purchase at `www.etsy.com/shop/RETROWORKSLLC`.

For example, with a shift of size 4, `SAGEHEN` becomes `WEKILIR`, and `MEET AT MIDNIGHT` becomes `QIIXDEXDQMHRMKLX`.

For now, let us lay out some functions that will help us to manage characters in lists and strings.

- **`Char.isAlpha`** `: char -> bool` returns true if the character is a letter.
- **`Char.isUpper`** `: char -> bool` returns true if the character is an uppercase letter.
- **`Char.isSpace`** `: char -> bool` returns true if the character is a blank space.
- **`Char.toUpper`** `: char -> char` changes a lowercase letter to uppercase, leaving all other characters unchanged.
- **`explode`** `: string -> char list` converts a string into the corresponding list of characters.
- **`implode`** `: char list -> string` converts a list of charecters into a string with the same characters.

Play with these functions a bit on the SML command-line to make sure you understand what they do. For example, typing `Char.isAlpha #","` yields the value `false`.

We want our encryption function to translate a string into a string, but it is easier to work with lists of characters. Therefore, we will write functions that operate on character lists. When we put it all together, the first step will be a call to `explode` and the last step will be a call to `implode`.

7. **[2 points]** It's cleanup time

The first step in encryption is to "clean up" the plaintext message. Write a function `sanitize` that takes a list of characters and returns a list from which all characters other than blanks and letters have been removed and in which all letters have been shifted to uppercase.

Assume that the plaintext string has already been exploded and work with a list of characters so that

```
implode (sanitize (explode "I recurse, therefore I am."))
```
yields `"I RECURSE THEREFORE I AM"`.

```
sanitize : char list -> char list
```

8. **[3 points]** Et tu, Brute?

Write a function `caesar` that takes an integer and a string and encodes the string using the shift specified by the integer. For example,

`caesar (7, "I recurse, therefore I am.")` yields `"PGYLJAYZLG OLYLMVYLGPGHT"`.

This is an exercise in piecing together functions that you have already written and ones that have already been mentioned. Present the sequential steps in some easy-to-understand way. Here is one possible structure:

The function `funPairs` from class may be helpful in carrying out the actual transformation of letters. use the question mark, `#"?"`, as the default character.

```
fun caesar (n, plainString) =
        let
            val plainList = ...
            ...
            val cipherList = ...
        in
            implode cipherList
        end;

caesar : int * string -> string
```