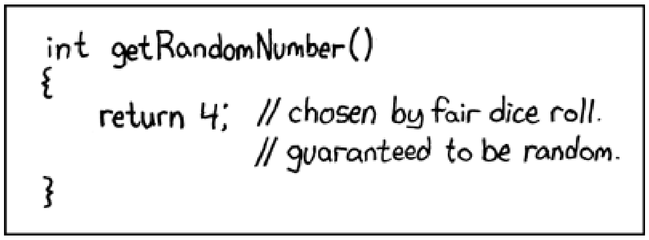
CS52 - Assignment 1

Due Friday 1/29 at 5:00pm



```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

`https://xkcd.com/221/`

For this assignment, you will start writing your first SML functions. Put all of your work in a file called `assign1.sml`, which should be placed in your `cs052` directory. In addition to your normal commenting, make sure to delimit problems by putting in a comment that has the problem number.

Reading

Make sure you're keeping up with the readings posted on the course web page.

Additionally, here are some relevant readings from some of the optional books:

- Ullman, Chapter 2, Sections 3.1 through 3.3, Section 5.1, and Sections 5.3 through 5.6;
- Gilmore, "Simple Applicative Programming";
- Hansen and Rischel, Chapters 2 and 5;
- Harper, parts of Chapters 2 through 9;
- Paulson, Chapter 1, Sections 2.1 through 2.6, Sections 2.14 through 2.16, and Sections 3.1 through 3.5; and
- Pucella, Chapter 2.

I don't expect you to read all of these things, but you should be trying to find a book that you like and get additional support/examples from there.

The Fun!

1. [**1 point**] Warming up

   Write a function `square` that computes the square of an integer.

   ```
   square : int -> int
   ```

2. [**1.5**] A little warmer

   (a) Write a function `factorial` to compute the factorial function.

   ```
   factorial : int -> int
   ```

   Your function must give the correct answer for all non-negative numbers and so something reasonable for negative numbers.

   (b) Experiment to find the largest n for which your function computes $n!$. Include in comments below the `factorial` function what that number is. (Do not include the experimental code in your file; simply the answer in a comment.)

   (c) `real`s have a much larger range of values than `int`s. One way to get the factorial function to calculate larger values is to still have it take in an `int` but then do all the calculations as a `real` and return a `real`. Write another function called `realFactorial` that returns a `real` result. You will need the built-in function `real` which converts an `int` into a `real`.

   ```
   realFactorial : int -> real
   ```

   Note: you should NOT simply have `real factorial x` as the body.

   (d) Repeat part b for the `realFactorial`.

3. [**1 point**] Takeuchi

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

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

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

4. [**1 point**] How long is that list?

   There is a built-in function `length` that returns the number of elements in a list. Pretend that `length` does not exist and write an equivalent function `myLength` that uses list recursion.

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

5. [**1 point**] How long is that string?

   There is a built-in function `size` that returns the number of characters in a string. Pretend that `size` does not exist and write an equivalent function `mySize`. (Hint: This is a one-liner. There is a built-in function `explode` that converts a string to a list of characters.)

   ```
   mySize : string -> int
   ```

6. [**1.5 points**] Let's put that function to use

   Write a function `squareAll` that takes a list of integers and returns the list of the squares of the respective integers. Use list recursion and your function `square` from Problem 1.

   ```
   squareAll : int list -> int list
   ```

7. [**1.5 points**] List recursion is your friend

   Write a function `sumList` that computes the sum of the numbers in a list of integers. (The sum of the numbers in the empty list is zero.)

   ```
   sumList : int list -> int
   ```

8. [**1 points**] Code reuse

   Use your functions from previous problems to write a function `sumSquares` that computes the sum of squares of the integers in a given list. (This is another one-liner. You may *not* use recursion!)

   ```
   sumSquares : int list -> int
   ```

9. [**1 points**] Simple counting

   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
   ```

   For example,

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

   Note: 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.

10. [**1.5 points**] Less simple counting

    The number of ways to pick m objects out of a set of n objects is given by the formula

$$\binom{n}{m} = \frac{n!}{m!\,(n-m)!}.$$

    (Do not be confused by the symbol $\binom{n}{m}$; it is the common mathematical notation for a particular function of m and n. It is sometimes written $C_n(m)$.)

    Write a function `comb0` to compute this expression directly, using your function `factorial` from Problem 2. Make `m` the first (leftmost) variable.

```
comb0 : int -> int -> int
```

11. [**2 points**] Bigger and better!

    The factorial function, and hence your function in the previous problem, is severely limited in the values it can handle (as you saw before!). An alternate way of calculating the function above that *does not use factorials at all* utilizes the following recursive identity:

$$\binom{n}{m} = \binom{n-1}{m-1} + \binom{n-1}{m}.$$

    Use this identity to write a *recursive* function called `comb1`.

```
comb1 : int -> int -> int
```

    You should not use any variants of your factorial function here (or *comb0*). Just implement the function recursively. The key challenge is to discover *all* of the base cases.

    Here is a justification of the identity: If I am choosing m items from n, I either take the first item of the n and then I have $m-1$ choices out of the remaining items, or else I skip the first item and I have m choices out of the remaining items.)

    An aside... Take a step back (good work so far!) and look back at the functions you've written so far. Identify the various strategies that you used to create functions:

    - Combining existing functions. No recursion is involved.
    - Using recursion on the natural numbers. The result is usually another integer.
    - Using list recursion. The result may be another list or a value that is not a list.

    A large part of our work in the early part of this course will be in extending and combining these basic ideas. The next problems provide an example of how to combine the two kinds of recursion.

12. [**3 points**] Getting more interesting

    (a) Write a function `consAll0` that prepends `0` to each element in a *list of lists of integers*. For example,

        `consAll0 [[1],[2,3]]` yields `[[0,1],[0,2,3]]`.

```
consAll0 : int list list -> int list list
```

(b) It does not really matter that we are prepending an integer. Write a more general (curried) function `consAllk` that takes a value `k` and a list of lists, and prepends `k` to each element in the list.

```
consAllk : 'a -> 'a list list -> 'a list list
```

13. [**3 points**] A bit more of a challenge

Using your solution to Problem 11 for inspiration and your function `consAllk` as an aid, write a function `combList` that will take an integer `k` and a list `xl` and return *a list* of all the `k`-element sublists of `xl`. For example,

`combList 2 [1,2,3]` yields `[[1,2], [1,3], [2,3]]`,
`combList 5 [1,2,3]` yields `[]`, and
`combList 0 [1,2,3]` yields `[[]]`.

Notice from the signature that `combList` returns a list of lists.

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

Hint: the length of this list should be the same as the number returned by your `comb` functions.

When you're done

Double check the following things:

- Make sure your code compiles, i.e. "use assignX.sml" should not give an exception.
- Make sure that your functions match the specifications *exactly*, i.e. the names should be exactly as written (including casing) and make sure your function takes the appropriate number of parameters and is curried/uncurried appropriately. For example, the function in Problem 3 is uncurried, so it would be called as
  `tak(3, 6, 7);`
  while the curried function in Problem 10 would be called without parentheses as
  `comb0 4 7;`
  Curried functions may seem a little weird at first, but they end up being easy and useful to use!
- Make sure you have used proper style and formatting. See the course readings for more information on this, but here are a few things to watch out for:
  - Be informative and consistent with your formatting!
  - No line should be longer than 80 characters.
  - You should use spaces (NOT tabs) in your submissions. If you're working on your own laptop, make sure the text editor you're using uses spaces instead of tabs.

- Make sure you've properly commented your code. You should include:
  - A comment header at the top of the file with your name, the date, the assignment number, etc.
  - Each problem should be delimited by comment stating the problem number.
  - Each function should have a comment above it explaining what the function does.
  - Complicating or unusual lines in functions should also be commented.

  Don't go overboard with commenting, but do be conscientious about it.

When you're ready to submit, upload your assignment via the online submission mechanism. You may submit as many times as you'd like up until the deadline. We will only grade the most recent submission.

Grading

functions	20
comments/style	2
Total	21.5