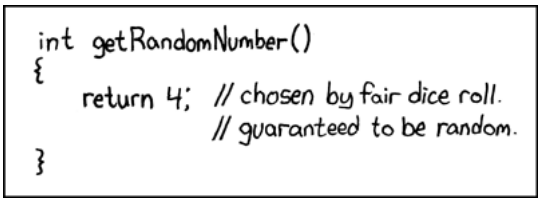
Computer Science 54

Assignment 1

Sunday, September 11 @ 11:59pm



https://xkcd.com/221/

This assignment consists of some warm-up exercises to give you more comfort with the systems we will be using. Work in your `cs54` directory. Copy the template file `assign1-template.sml` from the course web page, rename it as `assign1.sml`, and put your solutions in the places indicated. (Take a look at Assignment 0 if you need a refresher.) Copy the check file `assign1-check.sml` at the same time. If a problem asks for a solution that is not executable code—a number or explanation, for example—put the solution in a comment.

If you haven't yet, please read "An SML Style Guide," Part III of *A Brief Introduction to SML,* on the course page and follow its specifications and guidelines as you write your functions.

Signatures Please write your functions so that their names and signatures exactly match the specifications in the problem statements. Most of the specifications call for curried functions, so a function that takes three arguments would have a signature like this:

```
myCurriedSolution : int -> int -> int -> string
```

and be called like this:

```
myCurriedSolution 3 6 7;
```

Curried functions may seem a little unnatural at first, but they end up being easy and useful.

The alternative is an uncurried function which would have a signature like this:

Be sure to recognize that the symbols -> and * have significantly *different* meanings.

```
myUncurriedSolution : int * int * int -> string
```

and be called like this:

```
myUncurriedSolution (3, 6, 7);
```

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.

I Functions and Recursion on the Natural Numbers

0. **[0 points]** Not required (but not a bad idea)

Come by one of the instructors' offices, introduce yourself and tell them something interesting about yourself.

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

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

```
cube : int -> int
```

2. **[1 point]** A littler warmer

Write a function `min3` that takes a triple of integers (a three-tuple, not a list!) and returns the smallest of the three. For example, `min3 (29,183,47)` returns `29`.

```
min3 : int * int * int -> int
```

Take a few minutes to think about the logic so that you do not make it more complicated than it needs to be. Remember that the `if-then-else` construction represents a value.

3. **[2 points]** Now we're getting into it

a. Write a recursive function `factorial` to compute the factorial function. (Remember that 0! = 1. Have your function return 0 if the input is negative.)

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

b. Experiment to find the largest n for which `factorial` computes $n!$. Do not include the experimental code in your file; simply place the answer, as a constant, in a declaration for `maxFactorial`. Here is a sample with much too large a value.

```
val maxFactorial = 367;
```

c. Reals have a much larger range of values than integers. Write another function called `realFactorial` that takes an `int` argument, converts it into a `real`, carries out all the calculations with type `real`, and returns a `real` result.

You will need to use the built-in function `real` to convert the `int` argument into a `real`.

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

d. Experiment to find the largest n for which `realFactorial` computes a value for $n!$ without error. Expect to see a considerably larger value than you got in part b. Place your answer in a declaration for `maxRealFactorial`.

```
val maxRealFactorial : int
```

You should be able to calculate larger factorials using `realFactorial`. If your maximum value is the same or only slightly greater than that for `factorial`, you are probably still doing your calculations with integers. In that case, take another look at `realFactorial`.

II Recursion on Lists

In the previous exercises we considered recursion on the natural numbers. Now we turn to list recursion. In this part, you will write your own versions of some of SML's built-in list operations.

You are restricted to using only the following features of the language:

the constant `nil` or `[]`
the cons operator `::`
pattern-matching in function definitions
`if-then-else` constructions when absolutely necessary

Notice that the list does not include the append operator `@`. Be sure that you understand the difference between `::` and `@`.

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

Write a function `myLength` which counts the number of elements in a list.

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

5. **[1 point]** Let's put that function to use

Write a function `cubeAll` that finds the cube of every element in a list of integers. For example, `cubeAll [1,2,3]` returns `[1,8,27]`. You are encouraged to use the function `cube` that you wrote in Problem 1.

```
cubeAll : int list -> int list
```

6. **[1 point]** Double the fun

Write a function `duplicate` that duplicates each element of a list. For example, `duplicate [1,7,3]` returns `[1,1,7,7,3,3]`.

```
duplicate : 'a list -> 'a list
```

Notice that `duplicate` can be used on all kinds of lists, not just lists of integers.

7. **[1 point]** The end is in sight

Write a function `lessThanList` that takes as input a number and a list of numbers and returns `true` if the number is less than *all* of the numbers in the list, `false` otherwise. For example,

`lessThanList 10 [11, 47, 12]` returns `true`, but
`lessThanList 10 [11, 10, 47]` returns `false` (because `10` is not less than `10`).

Pay close attention to the base case of your recursion. The number `10` is indeed less than every element of the empty list.

```
lessThanList : int -> int list -> bool
```

8. **[1 point]** A bit more interesting

Write a function `myAppend` which appends its second argument onto the end of the first. You may *not* use the append (`@`) operator.

```
myAppend : 'a list -> 'a list -> 'a list
```

Hint: What should be the result of `myAppend [] [1, 2, 3]`?

The recursion will be carried out on only one of the two arguments. It is important to choose the appropriate one.