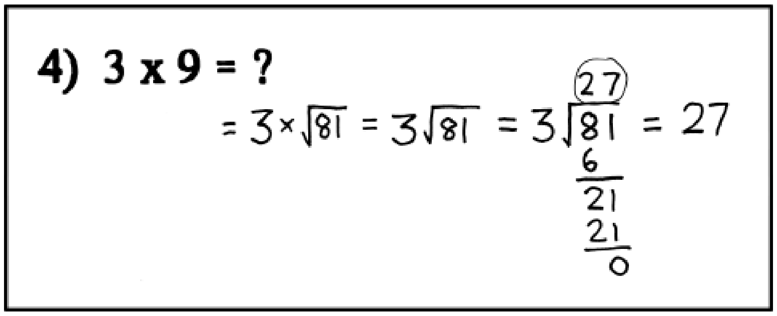
CS52 - Assignment 2

Due Friday 9/18 at 5:00pm



`https://xkcd.com/759/`

As mentioned in class, an important part of SML programming is recursion and particularly list recursion. For this assignment we'll be playing with these ideas even more!

Reading

Hopefully you have found one or more books that work for you. In your favorite SML book(s), finish reading the sections on lists and list recursion.

The Fun!

1. [**1 point**] Another path to success...

   In Assignment 1, you wrote functions `square` and `squareAll`. Use your `square` function and the built-in function `map` to write another version of `squareAll` that squares all the integers in a list (i.e. don't use recursion explicitly!).

2. [**2 points**] Recursion over two lists

   There is an SML function `ListPair.zip` that takes a pair of lists of the same lengths and forms a list of element-wise pairs. For example,

   `ListPair.zip ([1,2,3],[4,5,6])` yields `[(1,4),(2,5),(3,6)]`.

   When the two lists are of different lengths, the trailing elements of the longer list are ignored.

Use list recursion to implement directly a *curried* function `myZip` that has the same behavior. Do not use `ListPair.zip` in your function.

Hint: This problem is about the structure of list recursion. Think carefully about what constitutes a base case and how the recursive step reduces one case to a simpler one.

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

A fun excursion, but not part of the assignment: Search the web to find the documentation for `ListPair.zip` (We'll see more of these types of built-in functions as we progress through the course.)

3. [**2 points**]

   Write a function `split` that takes a list and produces an ordered pair of lists. The first element of the pair contains, in order, the first, third, etc. elements of the argument list, and the second element contains the elements at even positions. For example, `split [1,2,3,4,5]` yields `([1,3,5],[2,4])`.

   ```
   split : 'a list -> 'a list * 'a list
   ```

   Hint: The usual list-recursive paradigm has two cases: an empty list and a non-empty one. Instead, use a structure with two base cases: an empty list and a list with only one element—so that the recursive step is done on a list with at least two elements.

4. [**2 points**] Can you really do that?

   Write a function `cartesian` that takes two lists and forms a list of all the ordered pairs, with one element from the first list and one from the second. For example, `cartesian [1,3,5] [2,4]` will return `[(1,2),(1,4),(3,2),(3,4),(5,2),(5,4)]`.

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

   Hint: You will need to recurse over both lists, but do it separately. Make your function recursive in the *first* list. Pass the second list to an auxiliary function that takes an element and a list and returns a list of pairs formed from the element and the members of the second list. Remember to use `let` blocks appropriately here if applicable.

5. [**2 points**] Don't overthink it!

   Read about the SML keyword `op` in your favorite reference. Using the `cartesian` function from Problem 4, write a curried function `addAllNew` that takes two lists of integers and adds each number in the first list to each number in the second. For example, `addAllNew [2,4] [3,5,7]` returns `[5,7,9,7,9,11]`. There may be duplicates in the result list.

   ```
   addAllNew : int list -> int list -> int list
   ```

6. [**4 points**] Change is a good thing

   This is the infamous *Change Problem.* Given a list of coin denominations and an amount of money, we want to list all the ways to make change in that amount using the specified coins. For example, there are two ways to make seven cents from nickels and pennies:

```
[ [1,1,1,1,1,1,1], [5,1,1] ]
```

Write an SML function `change` that takes a list of denominations and an amount and produces a list of all the ways to make change in that amount. You may assume that the amount is not negative and that the coin denominations are all positive.

```
change : int list -> int -> int list list
```

This exercise is intended to be an example of a *nonstandard* use of list recursion. Be guided by the following:

- Think about the base cases. What is the result when the amount is 0? What is the result when the list of coins is empty?
- Here is a strategy to reduce a general case to simpler ones: Let a be the amount and d be the first denomination on the list. There are two possibilities: either you do not use d at all and you have to make change from the other denominations, or else you use d at least once and you have to finish the job by making change in the amount of $a - d$, using the entire list including d.
- The order of coins in a single possibility is not relevant. "Two pennies and a nickel" is the same as "a nickel and two pennies." If you follow the previous suggestion, you will automatically avoid duplicate possibilities. When you are finished, observe that the coins appear in a possibility in the same order—perhaps with repetitions or omissions—as in the original list of coins.
- The possibilities themselves may appear in any order, depending on how you implement the strategy above.
- Use the type information as a guide to constructing the final result. Remember that `change` returns a list of lists.

7. [**3 points**] Nums to lists and back again

   We can think of a natural number as being represented by a list of digits, with the least significant digit at the front of the list. For example, 47 is represented by `[7,4]`. Notice that the representation is not unique; 47 is represented by `[7,4]` and also `[7,4,0,0]`. What is the shortest representation for zero?

   Write functions to convert back and forth between non-negative integers and lists of digits. Declare the exceptions `NegativeNumber` and `BadDigit`. Raise the first if `toDigitList` encounters a negative argument. Raise the second if the function `fromDigitList` finds a number that is not in the range from 0 to 9.

```
toDigitList : int -> int list
fromDigitList : int list -> int
```

   Hint: Horner's rule is a method for efficiently calculating polynomials. One corollary of Horner's rule is that you can view a multi-digit number $d_1d_2...d_n$ (e.g. 342 has $d_1 = 3$, $d_2 = 4$ and $d_3 = 2$) as:

$$d_1 d_2 ... d_n = d_n + 10 * (d_{n-1} + 10 * (d_{n-2} + ...10 * d_1))$$

e.g.

$$342 = 2 + 10 * (4 + 3 * 10)$$

8. [**2 points**] Elementary school math

   Write a curried function `addDigitList` that adds two numbers represented by the digit list in Problem 7. Use the algorithm that you learned in elementary school.

   ```
   addDigitList : int list -> int list -> int list
   ```

   Comments:

   (a) The strategy "Convert to `int`, use the arithmetic operations built into SML, and convert back to a list" is *not* something you learned in elementary school.

   (b) Think about handling the problem a digit at a time, starting with the lower order digits.x

   (c) You will need an auxiliary function to help you out. Notice that if you do it by hand, very quickly you end up in a situation where you are adding *three* numbers, the two digits plus the carry bit from the previous digit. Your auxiliary function can help mimic this situation.

9. [**2 points**] It seemed easier in elementary school

   Write a curried function `multDigitList` that will multiply two numbers represented by the digit list in Problem 7. Again, use the algorithm that you learned in elementary school.

   ```
   multDigitList : int list -> int list -> int list
   ```

   Comments:

   (a) Again, you may not convert the numbers to int and then multiply.

   (b) Work through a few problems by hand. This will both serve as a way to try and identify the recursion as well as for debugging examples.

   (c) When doing it by hand, you should notice that a subproblem of multiplication involves multiplying a number by a single digit. You should tackle this as a separate helper function.

   (d) Your previous functions over int lists will be useful!

   (e) Soon we will talk about division. Think about how long division can be described as a recursive process. (They did not tell you about that in elementary school!)

Just for fun!

This problem is not part of the current assignment. It will be on Assignment 3, but it does not hurt to be thinking about it sooner.

You've done addition and multiplication, let's think about division. Sit down (not at a computer!) and do a few long division examples by hand. Think about repetition as you do it, i.e. where are there steps that are being done repeatedly.

Once you've thought about it, write a function that will carry out division on numbers represented as in Problem 7. Your function should be named `divDigitList`, be curried, and divide the second argument by the first. Because you are doing *integer* division, you will have a quotient and a remainder. The result will be an ordered pair, quotient first and remainder second. For example, the call `divDigitList [3] [3,1]` returns `([4],[1])`.

```
divDigitList : int list -> int list -> int list * int list
```

When you're done

Double check the following things:

- 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.
- Make sure you have used proper style and formatting. See the course readings for more information on this. Be informative and consistent with your formatting!
- 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	2.5
Total	22.5