CSE1729 - Introduction to Programming | January 28, 2022

Problem Set 2

Honor code. As a student in 1729, you have pledged to uphold the **1729 honor code**. Specifically, you have pledged that any work you hand in for this assignment must represent your individual intellectual effort.

1. The game "FizzBuzz" is a common group game in which people takes turns counting with the following stipulation:

   - If a number is divisible by 3, you must say "fizz".
   - If a number is divisible by 5, you must say "buzz".
   - If a number is divisible by both 3 and 5, you must say "fizzbuzz".
   - If a number is neither divisible by 3 nor by 5, you must say the number.

   (a) Write a SCHEME function `(fizzbuzz x)` whose input is a number `x` and whose output is the appropriate thing to say (either the number or the correct string term).

   (b) Implement `(fizzbuzz2 x)` in an alternate way by composing the results of two separate functions `(fizz x)` and `(buzz x)`. These functions should return their respective terms should the value `x` meet the criteria. You can utilize the `string-append` function. Note that any string appended with the empty string, "", is just the original string. For example

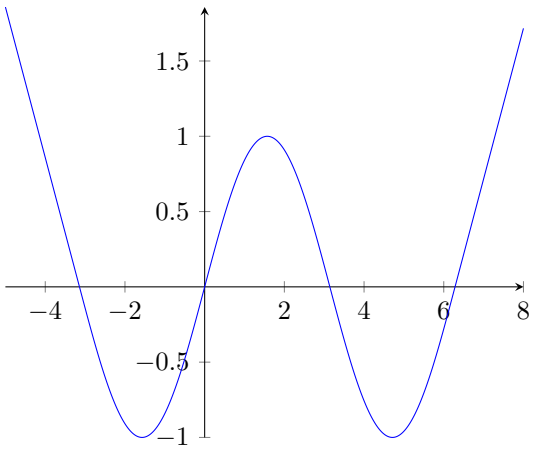
   ```
   >(string-append "Scheme is" "")
   "Scheme is"
   ```

2. Consider the following function

$$f(x) = \begin{cases} x - 2\pi & \text{if } x > 2\pi, \\ \sin(x) & \text{if } -\pi \leq x \leq 2\pi, \\ -x - \pi & \text{if } x < -\pi. \end{cases}$$

   Define a function `piecewise` so that `(piecewise x)` returns $f(x)$. In your code, please simply approximate π with the decimal 3.142. Use the built-in function `(sin x)` which returns $\sin(x)$.

   To visualize the function, the plot below shows the values in a region around zero.



3. Vi Hart's video explains, well, elementary algebra, which will be useful while solving this problem.

   Consider that the + and - builtin functions of scheme are broken and that you cannot use them. The only mathematical operations that work normally are the usual comparisons and boolean operators (i.e., `and, or, not` as well as $<, \leq, >, \geq, =$ all work fine). Your task is to rebuild addition (the same as the builtin +. Thankfully, this "broken" version of SCHEME has two working functions to increment an integer by 1 and decrement an integer by 1. Both could be implemented with

```
(define (inc x) (+ x 1))
(define (dec x) (- x 1))
```

   Your task is to implement the function `add` such that, for instance, `(add 3 10)` produces the answer `13`. Your add function is expected to work on non-negative integers (namely, the naturals). Your function starts like

```
(define (add n m) ...)
```

   And can only make use of `inc` and `dec` to do any kind of arithmetic. Observe that the following simple identities

$$\texttt{(add n m)} = \begin{cases} (n-1)+(m+1) & \Leftrightarrow n > 0 \\ m & \Leftrightarrow n = 0 \end{cases}$$

   are true. Namely, if you adopt a set-based interpretation of natural numbers (a natural number is a bag of pebbles), then to combine two bags of pebbles, one simply needs to move pebbles from one bag to the other, one by one...

4. Now that you have an addition function working, your next task is to rebuild the equally broken * function of SCHEME that multiplies natural numbers (non-negative). Again, the same assumption apply and you can only make use of `inc,dec` and your newly defined `add`. You should implement a function `mult` such that, for instance, the form `(mult 7 3)` evaluates to 21. Your function starts like

```
(define (mult n m) ...)
```

   Note that, once again, simply arithmetic identities also hold in this case. Simply find them and the solution will pop into existence!

5. Armed with `mult`, you can now turn your attention to the exponentiation function and build a function `power` that raises a base b to the integral (and positive) power n. Namely, the form `(power b n)` outputs b^n. Your function starts like

```
(define (power b n) ...)
```

   Once again, you can only use `inc,dec` and your newly minted `add` and `mult`. Naturally, recall that

$$\begin{aligned} b^n &= b^{n-1} \cdot b \\ b^0 &= 1 \end{aligned}$$

6. There is more than one way to achieve an expected result. For instance, you can use different algebraic properties to achieve the same result. Recall that

$$\begin{aligned} b^n &= (b^{\left(\frac{n}{2}\right)})^2 \Leftrightarrow n \text{ is even} \\ b^n &= (b^{\lfloor \frac{n}{2} \rfloor})^2 \cdot b \Leftrightarrow n \text{ is odd} \\ b^0 &= 1 \end{aligned}$$

The beauty of this function lies in how much *faster* it is compared to the version above in Question 5. Once again, you can only use `inc`, `dec` and your newly minted `add` and `mult`. Note that to compute the floor (rounded down value), you can use the SCHEME function `floor`. For division, the builtin SCHEME operator `/` works fine. Write a SCHEME function, named `(raise x n)`, which uses this faster method.

From this point onward, your + and * SCHEME functions are working again, feel free to use them!

7. Another algorithm for multiplying numbers is referred to by many names, including Russian Peasant Multiplication. First start with the two numbers you would like to multiply. Through this process, you will halve one number and double the other. Ignore any remainder in the result of the halving. Repeat the process until the halved number is one. Then, the product of the original two numbers is equal to the sum of just the doubled numbers for which the corresponding halved number is odd. For example,

37×42

halving	doubling
37	42
~~18~~	~~84~~
9	168
~~4~~	~~336~~
~~2~~	~~672~~
1	1344

$$37 \times 42 = 42 + 168 + 1344 = 1554$$

You may find this Numberphile video on Russian Multiplication helpful in understanding the method. Define a SCHEME function, named `(pmult x y)`, that uses Russian Peasant Multiplication to determine the product of x and y. You may find the `floor` function useful when removing any remainder from the result of division by two.

8. Recall from class the definition of `number-sum`, which computes the sum of the first n numbers:

```
(define (number-sum n)
  (if (= n 0)
      0
      (+ n (number-sum (- n 1)))))
```

   1. Adapt the function so that it computes *the sum of even numbers less than or equal to* n. Call your function `sumEven`. When evaluated at 4, your function should return the sum $2 + 4 = 6$. On 10, it should output $2 + 4 + 6 + 8 + 10 = 30$.
   2. Now, write a function that computes *the sum of odd numbers less than or equal to* n. Call your function `sumOdd`. When evaluated at 4, your function should return the sum $1 + 3 = 4$. On 10, it should output $1 + 3 + 5 + 7 + 9 = 25$.

9. Write a recursive function that, given a positive integer k, returns the product

$$\underbrace{\left(1 - \frac{1}{2}\right)\left(1 - \frac{1}{3}\right)\cdots\left(1 - \frac{1}{k}\right)}_{k-1}.$$

Call your function `h-product`. The $k - 1$ with the underbrace indicates that there are $k - 1$ of these terms.

(Experiment with the results for some various values of k; this might suggest a simple non-recursive way to formulate this function. Please hand-in the natural recursive version, though.)

10. Consider the problem of determining how many divisors a positive integer has. For example:

    - The number 4 has three divisors: 1, 2, and 4;
    - The number 5 has two divisors: 1 and 5;
    - The number 10 has four divisors: 1, 2, 5, and 10.

    In this problem you will write a Scheme function `(divisors n)` that computes the number of divisors of a given number n.

    The first tool you will need is a way to figure out if a given whole number ℓ divides another whole number n evenly. We provide the code for this, which you can just use as-is in your solution (it involves a function that we haven't talked about in class yet):

```
(define (divides a b) (= 0 (modulo b a)))
```

    Once you have defined this function, `(divides a b)` will be `#t` if a divides b evenly, and `#f` if not. For example:

```
> (divides 2 4)
#t
> (divides 3 5)
#f
> (divides 6 3)
#f
```

    At first glance, the problem of defining `(divisors n)` appears a little challenging, because it's not at all obvious how to express `(divisors n)` in terms of, for example, `(divisors (- n 1))`; in particular, it's not really clear how to express this function recursively.

    To solve the problem, you need to introduce some new structure! Here's the idea. Focus, instead, on the function `(divisors-upto n k)` which computes the number of divisors n has between 1 and k (so it computes the number of divisors of n upto the value k). Now you will find that there is a straightforward way to compute `(divisors-upto n k)` in terms of `(divisors-upto n (- k 1))`. Specifically, notice that

$$\texttt{(divisors-upto n k)} = \begin{cases} 0 & \text{if } k = 0; \\ 0 & \text{if } n = 0; \\ 1 & \text{if } k = 1; \\ 1 + \texttt{(divisors-upto n (- k 1))} & \text{if } k \text{ divides } n; \\ \texttt{(divisors-upto n (- k 1))} & \text{if } k \text{ does not divide } n. \end{cases}$$

    Write the Scheme code for the function `divisors-upto`; notice then that you can define

```
(define (divisors n) (divisors-upto n n))
```

    In this case, we call `divisors-upto` a "helper" function. What did it do? It let us "re-structure" the problem we wish to solve in such a way that we can recursively decompose it.

11. In combinatorial mathematics, a derangement is a permutation of the elements of a set, such that no element appears in its original position. Suppose that a professor gave a test to 4 students - A, B, C, and D - and wants to let them grade each other's tests. Of course, no student should grade his or her own test. How many ways could the professor hand the tests back to the students for grading, such that no student received his or her own test back? Out of 24 possible permutations (4!) for handing back the tests,there are only 9 derangements: BADC,BCDA, BDAC,CADB, CDAB,CDBA,DABC,DCAB,DCBA.

In every other permutation of this 4-member set, at least one student gets his or her own test back.
Check out this Numberphile video on this subject. They are solving a different problem (i.e. what is the probability at least one student gets their test back). But, it is another explanation of the problem.

The subfactorial, usually denoted $!n$, represents the number of derangements for a group of n elements, with the starting values $!0 = 1$ and $!1 = 0$.
Write a SCHEME function, named `(subfact n)` where

$$\text{subfact}(n) = \begin{cases} 1 & \text{if } n = 0, \\ 0 & \text{if } n = 1, \\ (n-1)(\text{subfact}(n-1) + \text{subfact}(n-2)) & \text{if } n > 1 \end{cases}$$

12. Recall from high-school trigonometry the sin function: If T is a right triangle whose hypotenuse has length 1 and interior angles x and $\pi/2 - x$, $\cos(x)$ denotes the length of the edge adjacent the angle x (here x is measured in radians). You won't need any fancy trigonometry to solve this problem.

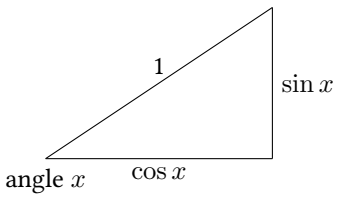


Figure 1: A right triangle.

It is a remarkable fact that for all real x,

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

Write a SCHEME function `new-cos` so that (new-cos x n) returns the sum of the first $(n+1)$ terms of this power series evaluated at x. Specifically, `(new-cos x 3)`, should return

$$1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!}$$

and, in general, `(new-cos x n)` should return

$$\sum_{k=0}^{n} (-1)^k \frac{x^{2k}}{(2k)!}.$$

You may use the built-in function `(expt x k)`, which returns x^k. It might make sense, also, to define `factorial` as a separate function for use inside your `new-cos` function.