CSE 1729 - Introduction to Principles of Programming — Due: September 9, 2019

Problem Set 1

1. Re-write the following arithmetic expressions as SCHEME expressions and show the result of the SCHEME interpreter when invoked on your expressions.
   (a) $(22+42)\times(54\times 99)$.
   (b) $((22+42)\times 54)\times 99$.
   (c) $64\times 102+16\times(44/22)$.
   (d) Is the expression in the following limerick, written by recreational mathematician James Mercer, correct?

   > A dozen, a gross, and a score
   > Plus three times the square root of four
   > Divided by seven
   > Plus five times eleven
   > Is nine squared and not a bit more

   That is, if you evaluate the following equation as a SCHEME expression with the logical operator $=$ for the equality symbol, does it evaluate to true (`#t`)?

   $$\frac{12+144+20+3\sqrt{4}}{7}+(5\times 11)=9^2+0$$

   Note: You can use the built-in SCHEME function `sqrt` to compute the square root of 4 as follows: `(sqrt 4)`

2. Reflect on the expressions above.
   (a) Of course, the first two expressions evaluate to the same number. In what sense are they different? How is this reflected in the SCHEME expression?
   (b) In an unparenthesized infix arithmetic expression, like $3+4*5$, we rely on a *convention* to determine which operation we apply first (rules of precedence). Are rules of precedence necessary for arithmetic operations SCHEME ?

3. Write SCHEME definitions for the functions below. Use the interpreter to try them out on a couple of test cases to check that they work, and include this output with your solutions.
   (a) `inc`, the function $\texttt{inc}(x)=x+1$.
   (b) `inc2`, the function $\texttt{inc2}(x)=x+2$. Show how to write `inc2` using your definition of `inc` instead of $+$.
   (c) `fifth`, the function $\texttt{fifth}(x)=x^5$.
   (d) `p`, the polynomial function $\texttt{p}(x)=(x^5+11x^4+27x^3+x+7)^2$.
   (e) Using the function `fifth`, write the function $\texttt{twenty-fifth}(x)=x^{25}$.
   (f) Using the function `twenty-fifth`, write the function $\texttt{six-hundred-and-twenty-fifth}(x)=x^{625}$.

Remark SCHEME provides built-in support for exponentiation (via the `expt` function, defined so that `(expt x y)` yields x^y). For the exercises above, however, please construct the functions $x \mapsto x^k$ using only $*$ and function application.

4. Reflect on your definition of `six-hundred-and-twenty-fifth` above. What would have been the difficulty of defining this merely in terms of $*$?

5. Write and test the following functions that deal with points and lines in the Cartesian plane.
   (a) `(y-value x b m)`, a function of three parameters (an x value, a y-intercept b, and a slope m) that returns the y value of the line at that x, that is $mx + b$.
   (b) `(points-slope x1 y1 x2 y2)`, a function of four parameters (the x and y values of two points) that calculates the slope of a line through those points (x_1, y_1) and (x_2, y_2). You may assume that the two points are distinct. (Note: this function is not required to work if the slope of the line is undefined.)
   (c) `(points-intercept x1 y1 x2 y2)`, a function of four parameters (the x and y values of two points) that calculates the y-intercept of a line through points (x_1, y_1) and (x_2, y_2). You may assume that the two points are distinct. (Note: this function is not required to work if the line does not have a y-intercept.)
   (d) `(on-parallels? x1 y1 x2 y2 x3 y3 x4 y4)`, a function of eight parameters (the x and y values of four points), returns true if the line through points (x_1, y_1) and (x_2, y_2) is parallel to the line through points (x_3, y_3) and (x_4, y_4). You may assume that points (x_1, y_1) and (x_2, y_2) are distinct, as are (x_3, y_3) and (x_4, y_4). (Note: this function can use any of the functions defined above, but should give a correct answer even in the cases where `points-slope` or `points-intercept` would fail.

6. Remember the *Quadratic formula*, which can be used to find the roots of a quadratic eguation? For a quadratic equation $ax^2 + bx + c = 0, a \neq 0$, the formula is

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

   Notice that this gives us two different roots (because of the $\pm$) whenever $b^2 - 4ac \neq 0$.

   Write the following SCHEME functions.
   (a) `(root1 a b c)` that gives us the root corresponding to the plus in the $\pm$ in the quadratic formula (that is, calculate $\frac{-b+\sqrt{b^2-4ac}}{2a}$).
   (b) `(root2 a b c)` that gives us the root corresponding to the minus in the $\pm$ in the quadratic formula (that is, calculate $\frac{-b-\sqrt{b^2-4ac}}{2a}$).
   (c) `(number-of-roots a b c)` which calculates the number of distinct roots to the equation $ax^2 + bx + c = 0, a \neq 0$ (which will either be 1 or 2).
   (d) `(real-roots? a b c)` is a boolean function that evaluates to `#t` when the roots of $ax^2 + bx + c = 0, a \neq 0$ are real numbers. Note that you do not have to calculate the roots to determine whether they are real or complex numbers.

   Note: there are some common calculations done in the above functions; it may make sense to write some auxiliary functions to make your code simpler.

7. **The Lazy Caterer's Sequence** describes the maximum number of pieces that can be made with n cuts of a pizza, pancake or any other circular item. For example, if all the cuts meet at a point (e.g. in the center) then 6 pieces can be made with 3 cuts (see Figure 1). However, the maximum number of pieces that can be made with three cuts is 7 (see Figure 2). The maximum number of pieces p that can be made with n cuts is given by:

$$p = \frac{n^2 + n + 2}{2}$$

Define a SCHEME function, named `(lazy n)`, which computes the maximum number of pieces that can be made from n cuts using the definition of the Lazy Caterer's Sequence above. The Lazy Caterer's Sequence begins (at 0):

$$1, 2, 4, 7, 11, 16, 22, 29, 37, 46, 56, 67, 79, 92, 106, 121, \ldots$$

Note, application of your function should evaluate to *one* of the numbers in the sequence.

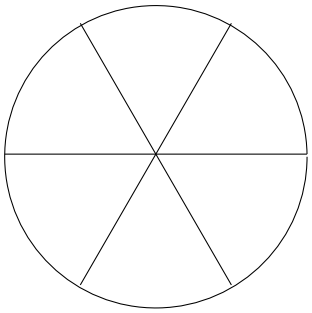


Figure 1: Pizza with three cuts and 6 pieces.

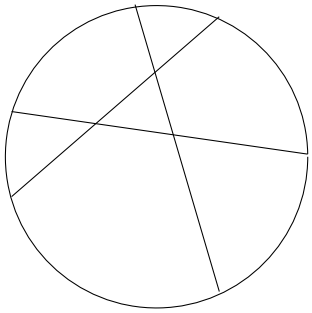


Figure 2: Pizza with three cuts and 7 pieces.

8. A cone is a three-dimensional geometric shape that tapers smoothly from a flat base (frequently, though not necessarily, circular) to a point called the apex or vertex. Figure 3 shows a cone with a circular base, referred to as a right circular cone.
   (a) Write a SCHEME expression to define a variable named pi for the constant π. You must use the value 3.14159 for your variable.
   (b) Define a SCHEME function, `(base r)` that takes the radius `r` of the circle that forms the base of a right circular cone and returns the *area* of the cone's base. You can, of course, just treat this as a circle. The area of a circle is πr^2.

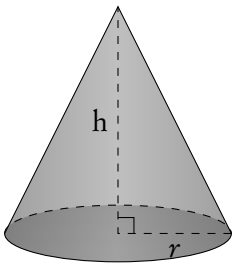


Figure 3: A right circular cone with height, h, and radius, r.

(c) Using your `base` function from part b, define a SCHEME function, `(cone-volume r h)`, which takes a the radius of the base of the cone, as well as the height of the cone as parameters and computes the volume of the cone. **You must use the base function in your solution.** The cone's volume is given by: $V = \frac{1}{3}bh$ where b is the area of the base and h the height from the base to the apex.

(d) The *lateral height* of a right circular cone is the distance from any point on the circle to the apex of the cone via a straight line along the surface of the cone. It is given by $\sqrt{r^2 + h^2}$ where r is the radius of the base the cone and h is the height. Define a SCHEME function, named `(lat-height r h)`, to compute the lateral height of a right circular cone.

(e) The lateral surface area of a right circular cone is $\pi r l$ where r is the radius of the circle at the bottom of the cone and l is the lateral height of the cone. The surface area of the bottom circle of a cone is the same as for any circle, πr^2. Thus, the total surface area of a right circular cone can be expressed as $\pi r^2 + \pi r\sqrt{r^2 + h^2}$. Define a SCHEME function, named `(cone-surface-area r h)`, to compute the total surface area of a cone. **You must use any functions written in previous parts to define `cone-surface-area`, if possible.**

CSE1729 - Introduction to Programming | September 6, 2019

Problem Set 2

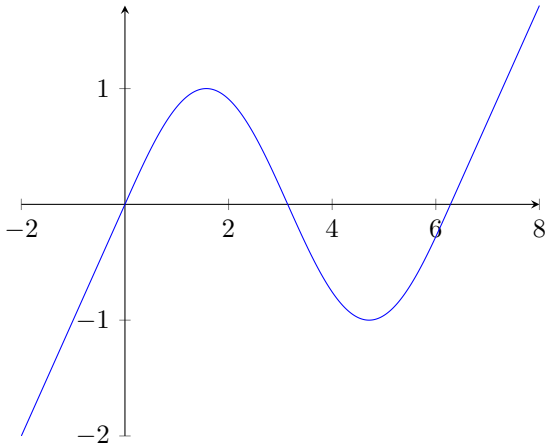
Honor code. As a student in 1729, you have pledged to uphold the **1729 honor code**. Specifically, you have pledged that any work you hand in for this assignment must represent your individual intellectual effort.

1. Consider the following function

$$f(x) = \begin{cases} x - 2\pi & \text{if } x > 2\pi, \\ \sin(x) & \text{if } 0 \leq x \leq 2\pi, \\ x & \text{if } x < 0. \end{cases}$$

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.



2. 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...

3. 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!

4. 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

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

5. 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 Q4. 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.

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

(a) 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 $2+4+6+8+10=30$.

(b) 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 $1+3+5+7+9=25$.

7. 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`.

(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.)

8. 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.

9. It is a remarkable fact that for all real x,

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

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

$$x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}$$

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

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

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-sin` function. (Aesthetic hint: Note that the value $2k$ is used several times in the definition of the kth term of this sum. Perhaps you can use a `let` statement to avoid computing this quantity more than once?)

10. In mathematics, Apéry's constant is defined as the number

$$\zeta(3) = \sum_{n=1}^{\infty} \frac{1}{n^3} = \frac{1}{1^3} + \frac{1}{2^3} + \frac{1}{3^3} + \frac{1}{4^3} + \cdots$$

where ζ is the Riemann zeta function. It has an approximate value of $\zeta(3) = 1.2020569031\ldots$. This constant arises naturally in a number of physical problems, including in the second- and third-order terms of the electron's gyromagnetic ratio using quantum electrodynamics. But, you don't need to know anything about that to solve this problem.

Give a SCHEME function, named `(apery k)` that evaluates to an approximation for Apéry's constant using k terms.

CSE1729 - Introduction to Programming	September 13, 2019
Problem Set 3	

Remark 1. *The Racket interpreter can maintain two different representations of numeric quantities: fractions and decimal representations. While fractions always represent exact numeric quantities, decimal expansions maintain a finite number of digits to the right of the decimal point. The Racket interpreter will attempt to infer, when dealing with numbers, whether to maintain them as fractions or as decimal expansions. For example*

```
> (/ 1 2)
1/2
> (/ 1 2.0)
0.5
> (+ (/ 1 2) (/ 1 3) (/ 1 6))
1
> (+ (/ 1 2) (/ 1 3.0) (/ 1 6)
0.9999999999999999
>
```

Clearly, the use of fractions (rationals) is more precise because of the rounding errors that occur with decimals. In general, the interpreter will maintain exact expressions for numeric quantities "as long as possible," expressing them as fractions. You can instruct the interpreter that a number is to be treated as a decimal by including a decimal point: thus `1` *is treated as an exact numeric quantity, whereas* `1.0` *is treated as a decimal expansion. The interpreter knows how to convert fractions to decimals (you can configure the number of digits of accuracy you wish), but will never convert decimals back to fractions (or integers). (So you know, this process is called* type-casting*.). You can "force" a conversion from a fraction to a decimal easily. Indeed, writing* `(+ (/ 1 3) 0.0)` *or even* `(exact->inexact (/ 1 3))` *will produce a decimal from the fraction* $\frac{1}{3}$.

Arithmetic expressions like `(+ 1 1.0)` *pose a problem because the two arguments are of different "types." In this case, the interpreter will transform the exact argument* (`1`) *into a decimal value, and then proceed as though all arguments were decimals (returning a decimal result). Other arithmetic operations are treated similarly.*

1. Define $H_n = \frac{1}{1} + \frac{1}{2} + \ldots + \frac{1}{n}$; these are referred to as the *harmonic numbers*. A remarkable fact about these numbers is that as n increases, they turn out to be very close to $\ln n$. ($\ln n$ is the *natural logarithm* of n.) In particular, as n increases the difference $|H_n - \ln n|$ converges to a constant (Euler's constant).

   (a) Give a SCHEME function, called `harmonic` so that `(harmonic n)` returns H_n.

   (b) Using your function from the previous part, give an estimate of Euler's constant. Specifically, write a SCHEME function `Eulerest` so that `(Eulerest n)` returns the absolute value of the difference between H_n and $\ln n$. (You may wish to use the SCHEME function `(log x)` which returns the natural logarithm of x.) So you know you are in the ballpark, Euler's constant is a little over a half.

2. A integer $n > 1$ is prime if its only positive divisors are 1 and n. (The convention is not to call 1 prime.) The following SCHEME procedure determines if a number is prime. Note how it uses a nested definition `divisor?` (see class) to keep the code clean and tidy.

```
(define (prime? n)
  (define (divisor? k) (= 0 (modulo n k)))
  (define (divisors-upto k)
    (and (> k 1)
         (or (divisor? k) (divisors-upto (- k 1)))))
  (not (divisors-upto (- n 1))))
```

(So, it returns `#t` for prime numbers like 2, 3, 5, 7, and 11 and `#f` for composite (that is, non-prime) numbers like 4, 6, 8, and 9.) Using this procedure, write a function `count-primes` so that `(count-primes m)` returns the number of prime numbers between 1 and m.

3. The Lucas numbers are a sequence of integers, named after Édouard Lucas, which are closely related to the Fibonacci sequence. In fact, they are defined in very much the same way:

$$L_n = \begin{cases} 2 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ L_{n-1} + L_{n-2} & \text{if } n > 1. \end{cases}$$

   (a) Using the recursive description above, define a SCHEME function `Lucas` which takes one parameter, n, and computes the n^{th} Lucas number L_n.

   (b) The small change in the "base" case when $n = 0$ seems to make a big difference: compare the first few Lucas numbers with the first few Fibonacci numbers. As you can see, the Lucas numbers are larger, which makes sense, and—in fact—the difference between the nth Lucas number and the nth Fibonacci number grows as a function of n.

   Consider, however, the ratio of two adjacent Lucas numbers; specifically, define

$$\ell_n = \frac{L_n}{L_{n-1}}.$$

   Write a SCHEME function `Lucas-ratio` that computes ℓ_n (given n as a parameter). Compute a few ratios like ℓ_{20}, ℓ_{21}, ℓ_{22}, ...; what do you notice? (It might be helpful to convince SCHEME to print out the numbers as regular decimal expansions. One way to do that is to add `0.0` to the numbers.)

   Now define

$$f_n = \frac{F_n}{F_{n-1}}$$

   where F_n are the Fibonacci numbers. As above, write a SCHEME function `Fibonacci-ratio` to compute f_n and use it to compute a few ratios like f_{20}, f_{21}, f_{22}, ...; what do you notice?

   (c) *Computing with a promise.* Ask your SCHEME interpreter to compute L_{30}, then L_{35}, then L_{40}. What would you suspect to happen if you asked it to compute L_{50}?

   Consider the following SCHEME code for a function of four parameters called `fast-Lucas-help`. The function call

```
(fast-Lucas-help n k lucas-a lucas-b)
```

   is supposed to return the nth Lucas number *under the promise that it is provided with any pair of previous Lucas numbers*. Specifically, if it is given a number $k \leq n$ and the two Lucas number L_k and L_{k-1} (in the parameters `lucas-a` and `lucas-b`), it will compute L_n. The idea is this: If it was given L_n and L_{n-1} (so that $k = n$), then it simply returns L_n, which is what is was supposed to compute. Otherwise assume $k < n$, in which case it knows L_k and L_{k-1} and wishes to make some "progress" towards the previous case; to do that, it calls `fast-Lucas-help`, but provides L_{k+1} and L_k (which it can compute easily from L_k and L_{k-1}). The code itself:

```
(define (fast-Lucas-help n k luc-a luc-b)
  (if (= n k)
      luc-a
      (fast-Lucas-help n (+ k 1) (+ luc-a luc-b) luc-a)))
```

   With this, you can define the function `fast-Lucas` as follows:

```
(define (fast-Lucas n) (fast-Lucas-help n 1 1 2))
```

(After all, $L_0 = 2$ and $L_1 = 1$.)

Enter this code into your SCHEME interpreter. First check that `fast-Lucas` agrees with your previous recursive implementation (`Lucas`) of the Lucas numbers (on, say, $n = 3, 4, 5, 6$). Now evaluate `(fast-Lucas 50)` or `(fast-Lucas 50000)`.

There seems to be something qualitatively different between these two implementations. To explain it, consider a call to `(Lucas k)`; how many total recursive calls does this generate to the function `Lucas` for $k = 3, 4, 5, 6$? Now consider the call to `(fast-Lucas-help k 1 1 2)`; how many recursive calls does this generate to `fast-Lucas-help` for $k = 3, 4, 5, 6$? Specifically, populate the following table (values for $k = 1, 2$ have been filled-in):

	Recursive calls made by `(Lucas k)`	Recursive calls made by `(fast-Lucas-help k 1 1 2)`
$k = 1$	0	0
$k = 2$	2	1
$k = 3$		
$k = 4$		
$k = 5$		
$k = 6$		

4. In mathematics and other fields, two quantities $a > b$ are said to have the *golden ratio* if

$$\frac{a+b}{a} = \frac{a}{b}.$$

For example, the heights of adjacent floors of Notre Dame cathedral are in this proportion, as are the spacings of the turrets; the side-lengths of the rectangular area of the Parthenon have this ratio.

Mathematically, it is easy to check that if a and b have this relationship then the ratio $\phi = a/b$ is the unique positive root of the equation $\phi + 1 = \phi^2$, which is approximately 1.618.

(a) The golden ratio also satisfies the equation $\phi = 1 + \frac{1}{\phi}$. This formula can be expanded recursively to the *continued fraction*:

$$\phi = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \ddots}}$$

Define a recursive SCHEME function which takes one parameter, n, and computes an approximation to the value of this repeated fraction by expanding it to depth n. To be precise, define

$$\Phi_1 = 1 + \frac{1}{1} = 2,$$
$$\Phi_2 = 1 + \frac{1}{1 + \frac{1}{1}} = \frac{3}{2},$$
$$\Phi_3 = 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1}}} = 1\frac{2}{3},$$
$$\ldots$$

or, more elegantly,

$$\Phi_1 = 2,$$
$$\Phi_n = 1 + \frac{1}{\Phi_{n-1}} \quad \text{for } n > 1.$$

Your function, called `golden`, when given n, should return Φ_n.

(b) As mentioned above, the golden ratio satisfies the equation $\phi^2 = 1 + \phi$. This can be expanded into a different recursive formula that yields a *"continued square root"*:

$$\phi = \sqrt{1 + \sqrt{1 + \sqrt{1 + \sqrt{1 + \cdots}}}}$$

Define a SCHEME function that takes one parameter, n, and computes the n^{th} convergent of the golden ratio using the continued square root. To be more precise, define:

$$\Phi_n = \begin{cases} 1 & \text{if } n = 0, \\ \sqrt{1 + \Phi_{n-1}} & \text{if } n > 0. \end{cases}$$

(Note that these Φ_i are different from those in the previous part of the problem.) Now define a SCHEME function `golden-sqrt` so that `(golden-sqrt n)` returns the value Φ_n. (Use the SCHEME function `(sqrt x)` which returns the square root of x.)

5. Consider the problem of defining a function `interval-sum` so that `(interval-sum m n)` returns the sum of all the integers between m and n. (So, for example `(interval-sum 10 12)` should return the value $33 = 10 + 11 + 12$.) One strategy is

```
(define (interval-sum m n)
  (if (= m n)
      m
      (+ n (interval-sum m (- n 1)))))
```

and another solution, which recurses the "other direction" is

```
(define (interval-sum m n)
  (if (= m n)
      m
      (+ m (interval-sum (+ m 1) n))))
```

These both work. It seems like one should be able to combine these to produce another version:

```
(define (interval-sum m n)
  (if (= m n)
      m
      (+ m
         (interval-sum (+ m 1) (- n 1))
         n)))
```

But this only works for certain pairs of input numbers. What's going on?

6. Define a SCHEME procedure `(mth-summation n m)` to compute the m^{th} summation of the first n natural numbers. That is,

$$sum(n, m) = \begin{cases} 0 & \text{if } n = 0, \\ n + sum(n-1, 1) & \text{if } m = 1, \\ sum(sum(n, m-1), 1) & \text{otherwise.} \end{cases}$$

7. The famous Ackermann Function, named for Wilhelm Ackermann whose doctoral advisor was David Hilbert (a famous mathematician) is generally expressed today as:

$$\text{ack}(m,n) = \begin{cases} n+1 & \text{if } m = 0, \\ \text{ack}(m-1,1) & \text{if } m > 0 \text{ and } n = 0, \\ \text{ack}(m-1,\text{ack}(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

What makes this function absolutely recursive is the second parameter to the recursive call to ack in the last case. The second parameter to ack in this case is itself a recursive call to ack. Notice, that in each recursive call, either m or n are decreased and neither are ever increased. Therefore, this algorithm will always terminate, eventually. Because of the diabolical recursive second parameter to the recursive call in the last case, computing this function for any values other than fairly small values of m and n takes more time to compute than any reasonable human being would wait. In fact, computing ack$(4,1)$ took longer than three minutes on my laptop. Note, however, that even though we can not realistically compute Ackermann's function for fairly large values for m and n, we do know that if we could build a computer that would last long enough and could store extremely large integer values, the algorithm would eventually terminate with the solution.

Define a SCHEME function, named `(ack m n)`, that computes the Ackermann function as defined above. For testing, your function should be able to compute the following:

```
>(ack 3 4)
125
>(ack 4 0)
13
```

8. The persistence of a number is the number of times you must apply an operation to that number until the result of applying that operation no longer changes the number. For this problem, you will count the number of times one must multiply the digits of a positive integer until the result is the same as the integer.

   For example, the multiplicative persistence of the number 39 is 3 because $3 \cdot 9 = 27$, $2 \cdot 7 = 14$, and $1 \cdot 4 = 4$. You stop at 4 because multiplying 4 (by the multiplicative identity, which is 1) equals 4. Try it in the SCHEME interpreter.

   Define a SCHEME procedure, named `(mult-persistence n)`, which accepts a positive integer, n, and evaluates to the multiplicative persistence of n.

1. In a certain school, there are 1,000 lockers, numbered 1, 2, . . . , 1,000. Happily, there are also 1,000 students. When the first student enters the building in the morning, she opens each locker as she walks by. Simple enough. When the second student enters the building, however, she decides to *toggle* the state of every *second* locker: to be precise, if 2 divides the name of the locker, she changes the locker's state (if was open, she closes it, if it was closed, she opens it). Similarly, when the 3rd student enters, she toggles the state of every third locker. This process continues, with the kth student toggling the state of all lockers whose name is evenly divisible by k. We'd like to determine the the state of the ℓth locker after all the students have entered the school and done their quixotic locker adjustments.

   Notice that locker ℓ can only touched by the kth student if k divides into ℓ evenly; in particular, once the first ℓ students have handled the lockers, the ℓth locker is in its final state—no further students will toggle it.

   To make sure you understand the problem, see that you agree that after all students have had their turn to toggle the lockers,

   (a) locker 1 is open,

   (b) locker 2 is closed,

   (c) locker 3 is closed,

   (d) locker 4 is open, and

   (e) locker 5 is closed.

   (a) Write a SCHEME function `(locker-open k)` that determines if the kth locker is open after all the students have had their turns. (It should return a Boolean value, `#t` if the locker is open and `#f` if the locker is closed.) (Hint: You might start by defining a function `(locker-state l k)` which computes the state of the ℓth locker after the first k students have made their pass on the lockers. If you do use such a subordinate function, be sure to hide its definition inside your definition of `locker-closed` so as not to clutter the containing environment.)

   (b) Write a SCHEME function `(find-open k)` that *prints* on the standard output the first k lockers that are open after all students entered the school. Follow your printout by the word "done". You can make use of the SCHEME function `(display f)` which prints the value of f on the standard output. For instance, the fragment:

   ```
   (display "hello ")
   (display  2)
   (display " you\n")
   ```

   prints on the standard output the string

   ```
   hello 2 you
   ```

   followed by a line feed (the '\n' character). You may, once again, find that a helper function `(enum-locker i n)` can be quite handy to print the next n lockers that happen to be open when starting from locker i. Naturally, do hide `enum-locker` within your implementation of `find-open`. For instance, the output for `(find-open 2)` should be

   ```
   1
   4
   done
   ```

2. The Gauss-Legendre algorithm is an algorithm to compute the digits of π. It is notable for being rapidly convergent, with only 25 iterations producing 45 million correct digits of π. The algorithm starts with the following initial values for four variables:

$$a_0 = 1 \qquad b_0 = \frac{1}{\sqrt{2}} \qquad t_0 = \frac{1}{4} \qquad p_0 = 1$$

Next, the following instructions are repeated until the difference of a_n and b_n are within the desired accuracy:

$$a_{n+1} = \frac{a_n + b_n}{2} \qquad b_{n+1} = \sqrt{a_n b_n}$$
$$t_{n+1} = t_n - p_n(a_n - a_{n+1})^2 \qquad p_{n+1} = 2p_n$$

Then, π is approximated by $\pi \approx \frac{(a_{n+1}+b_{n+1})^2}{4t_{n+1}}$

Define a Scheme function, named `(gauss-legendre tol)`, that takes one parameter representing the tolerance (i.e. the desired accuracy) and computes an approximation to π using the Gauss-Legendre algorithm. You may once again find that a helper function makes this easier. If you do, make sure it gets hidden within the implementation of `gauss-legendre`.

3. Consider the harmonic numbers $H_n = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$. Last week you wrote a recursive Scheme function (named `harmonic`) which, given a number n, computes H_n. Revise your `harmonic` function to take advantage of the `sum` function seen in class and shown below:

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

Of course, your new and improved definition should not be recursive and should rely on `sum` to do the hard work. Your new function should be called `harmonic` (and take one argument).

4. For two functions f and g and an integer $n \geq 0$ we are interested in the sum of fractions of the form:

$$\frac{f(-n)}{g(-n)} + \cdots + \frac{f(n)}{g(n)}.$$

(Note that this sum has $2n + 1$ terms in it, as it evaluates the fraction $f(x)/g(x)$ at all points between $-n$ and n.) Note that a little care is required here, because the quantity is not even defined if $g(k) = 0$ for one of the $k \in \{-n, \ldots, n\}$. To work around this issue, we define

$$R(f, g; n) = q(-n) + \cdots + q(n)$$

where

$$q(k) = \begin{cases} 0 & \text{if } g(k) = 0, \text{ and} \\ \frac{f(k)}{g(k)} & \text{otherwise.} \end{cases}$$

Write a Scheme program, called `frac-sum` so that `(frac-sum f g n)` returns the value $R(f, g; n)$ as defined above.

5. Recall the definition of the derivative of a function from calculus (you will not need to know any calculus to solve this problem):

$$f'(x) = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}$$

By choosing very small values for h in the above equation, we can get a good approximation of $f'(x)$.

(a) Write a function (call it `der` for *derivative*) in SCHEME that takes a function f and a value for h as formal parameters and returns the function g defined by the rule

$$g(x) = \frac{f(x+h) - f(x)}{h}.$$

As mentioned above, for small `h`, g is a good approximation for the derivative of f.

Important note: Your function should take a *function* and a *number* as arguments (for example `(der f h)` where `f` is a function and `h` is a number) and return a *function*.

(b) Now take it a step further and write a function which takes three formal parameters f, n and h and returns an approximation of the n^{th} derivative of f. Call your function `der-n`; then `(der-n f n h)` is an example call to the function. Make use of the derivative function you just wrote. (Specifically, you wish to return the function obtained by applying `der` to your function n times.)

6. *Newton's Method* is an iterative method for finding successively better approximations to the roots (that is, the zeroes) of a real-valued function. To be more precise, given a function f, Newton's Method is an approach to find a value x for which $f(x) \approx 0$. Newton's Method requires an initial guess for the root (x_0) and determines a sequence of values $x_1, x_2, \ldots$ defined by the recursive rule:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

For many functions of interest, these "iterates" converge very quickly to a root.

For example, consider the polynomial $p(x) = x^2 - x - 1$. It turns out that the positive root of p is the number 1.618... that you computed in many ways on the last problem set (the golden ratio). Let's see Newton's method in action. It turns out that the derivative of p is the polynomial $p'(x) = 2x - 1$. (You don't need to know how to determine the explicit derivative of functions for this problem—you'll be using instead the code you generated in the last problem.) Starting with the guess $x_0 = 2$, we may run Newton's method forward to find that:

$$\begin{aligned}
x_0 &= 2 \\
x_1 &= x_0 - \frac{p(x_0)}{p'(x_0)} = 2 - \frac{p(2)}{p'(2)} = \frac{5}{3} = 1.666\ldots \\
x_2 &= x_1 - \frac{p(x_1)}{p'(x_1)} = \frac{5}{3} - \frac{p(5/3)}{p'(5/3)} = 1\tfrac{13}{21} = 1.61904\ldots \\
&\ldots
\end{aligned}$$

Write a SCHEME function that implements Newton's Method–call it `newton`. Your function should accept three formal parameters, a function f, an initial guess for the root x_0 and the number of iterations to run, n. You can make use of the derivative function from the previous problem (with `h` set to .01).

Specifically, the call `(newton f x n)` should return the nth iterate, as defined above, of Newton's method. (Thus `(newton f x 0)` should return x.)

To test your implementation, you can try the following in DrRacket.

(a) Use your implementation can find the root of the function $f(x) = 2x^2 - 1$. (start with an initial guess of 4).

(b) Use your solution to find a good approximation to the golden ratio by working with the polynomial $g(x) = x^2 - x - 1$ (start with the guess 2).

7. SICP, Problem 1.29 (Simpson's rule). Recall that the *integral* of a function f between a and b (with $a < b$) is the area underneath the function on the interval $[a, b]$. See Figure 1. Simpson's rule, described in your book, is a method for *approximating* this value.

To begin with, define a function `(sum-term term a b)` that takes a function `term` and two integers a and b as arguments and returns the sum $\texttt{term}(a) + \texttt{term}(a+1) + \cdots + \texttt{term}(b)$. Use this in your solution by defining a function that computes the Simpson's rule terms and passing this to `sum-term`.

Once `sum-term` is defined, as above, use this to construct a function `simpson-integrate` so that

`(simpson-integrate f a b n)`

returns the estimate to the integral of the function f, on the interval $[a, b]$ with n samples, according to Simpson's rule. Recall that Simpson's rule is simply:

$$\int_a^b f(x)dx = \frac{\Delta x}{3} \cdot (f(x_0) + 4 \cdot f(x_1) + 2 \cdot f(x_2) + 4 \cdot f(x_3) + 2 \cdot f(x_4) + \ldots + 4 \cdot f(x_{n-1}) + f(x_n))$$

where $\Delta x = \frac{b-a}{n}$ and $x_i = a + i \cdot \Delta x$.

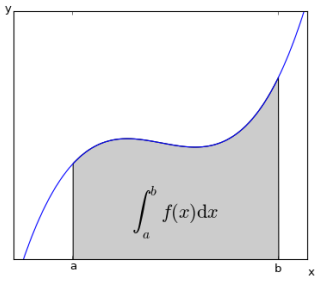


Figure 1: The integral of f from a to b is the area of the shaded region above.

To check your solution, you might try integrating the functions $f_1(x) = x$, $f_2(x) = x^2$, and $f_4(x) = x^4$ on the interval $[0, 1]$ (so $a = 0$ and $b = 1$). You should find that the integral of f_1 is $1/2$ (the area of the triangle), the integral of f_2 is $1/3$, and the integral of f_4 is $1/5$.

8. Adapt your Simpson's rule integrator so that the computation is tail-recursive. All that is necessary is to rewrite your `sum` function so that it is tail-recursive (cf. SICP Exercise 1.30.). Specifically, name your function `(new-sum-term term a b)` where *term* is the univariate function and a and b give the bounds of the range to accumulate.

CSE1729 - Introduction to Programming — October 11, 2019

Problem Set 5

1. item Define a SCHEME function, `zip`, which takes as arguments two lists $(a_1 \dots a_n)$ and $(b_1 \dots b_n)$ of the same length, and produces the list of pairs $((a_1.b_1) \dots (a_n.b_n))$.

2. Define a SCHEME function, `unzip`, which takes a list of pairs $((a_1.b_1) \dots (a_n.b_n))$ and returns a *pair* consisting of the two lists $(a_1 \dots a_n)$ and $(b_1 \dots b_n)$. Note that these functions are not exactly inverses of each other, since `zip` takes its lists as a two arguments, while `unzip` produces a pair of lists. Finally, it is wise to realize how SCHEME prints out a pair of lists. Consider the statement `(cons (list 1 2) (list 3 4))` which denotes a pair of lists. It prints as `((1 2) 3 4)`.

3. Consider again the problem of *making change*. Given a list of "denominations" $(d_1 d_2 \dots d_k)$ values and a positive integer n, the problem is to determine the number of ways that n can be written as a sum of the d_i. For example, if we consider US coins, (1 5 10 25), the number 11 can be written in 4 ways:

$$10+1,\ 5+5+1,\ 5+\underbrace{1+\cdots+1}_{6},\quad \text{and}\quad \underbrace{1+\cdots+1}_{11}.$$

   Note that we do not consider $10+1$ and $1+10$ as "different" ways to write 11: all that matters is the number of occurrences of each denomination, not their order. Write a SCHEME function `(change k ℓ)` that returns the number of ways to express `k` as a sum of the denominations appearing in the list ℓ. This is a generalization of the version seen in class early on this semester.

4. Now that you can *count* the number of ways to give change, it is time to compute each way to give change! Write a SCHEME function `(make-change n den)` which outputs a list of change specifications. Namely, a call `(make-change 11 (list 25 10 5 1))` produces the list

```
((1 1 1 1 1 1 1 1 1 1 1) (1 1 1 1 1 1 5) (1 5 5) (1 10))
```

   Hints: a helper function `(helper n den cur)` that takes as input *cur*, a list of coins given out so far will surely come in handy! Also, the order in which the "options" appear in the top list is immaterial.

5. While getting the output above is helpful, it is not particularly readable. Indeed, the list `(1 1 1 1 1 1 5)` that states 6 pennies and 1 nickel could be far more readable as `((6 . 1) (1 . 5))` also stating 6 pennies and 1 nickel. That is, this is a list of pairs telling us how many of each denomination. Thankfully, the former can be translated into the latter by a conversion known as *run length encoding* that replaces sequences of an identical value by a pair giving the number of repetition of the value. Write a SCHEME function `(rle coins)` which, given a list of coins, returns a list of pairs denoting the number of repetitions of each sequence of consecutive values. As a last example, the list

```
(list 1 1 1 1 1 1 1 5 5 5 5 1 1 1 10 10 10 1 1 25 25 25 25 25 25)
```

   would be encoded as

```
((7 . 1) (4 . 5) (3 . 1) (3 . 10) (2 . 1) (6 . 25))
```

   Note how the tree sub-sequences of pennies are **not** collapsed into a single value. Those are kept as separate pairs. Naturally, this function only works for one element of the output from `make-change`.

6. Naturally, it would be nice to convert the entire output of `make-change` to the RLE format. Write a SCHEME function `(rle-all lcoins)` which, when given a list of coin changes, produces a list of RLE encoded coin-changes. For instance the call

```
(rle-all (make-change 11 '(25 10 5 1)))
```

produces

```
(((11 . 1)) ((6 . 1) (1 . 5)) ((1 . 1) (2 . 5)) ((1 . 1) (1 . 10)))
```

which is a list of 4 lists (since there are four ways to give change on 11 cents) where each list is an RLE encoding.

7. Write a SCHEME function `positives` which takes a list—call it ℓ—as an argument and returns a list consisting of all elements of ℓ that are positive. In particular, once you have `positives` defined correctly, you should be able to reproduce the following behavior.

```
> (positives (list -2 -1 0 1 2))
'(1 2)
> (positives (list 2 1 0 -2 -1))
'(2 1)
> (positives '(3 1 -1 1 -1))
'(3 1 1)
```

To keep things simple, it's fine if your function just removes from the list all numbers that are zero or less (leaving duplicates in the remaining list, as shown above).

8. Write a SCHEME function `remove-duplicates` that removes all duplicates from a list. (**Hint:** you might start by defining a function which removes all duplicates of a particular given value v from a list; then what?)

So, for example,

```
> (remove-duplicates '(1 2 3 4 5 3 1))
'(1 2 3 4 5)
```

CSE1729	Introduction to Programming Problem Set 6	October 18, 2019

1. *Pearson's Coefficient* is widely used in the natural sciences as a measure of "correlation" between two variables. For example, it is reasonable to expect that human height and weight are correlated, which is to say that—on average—taller humans tend to weigh more. As an example, consider the heights and weights of a small population of humans, as shown in the table below:

	A	B	C	D	E	F	G
height (cm)	160	186	172	202	177	186	191
weight (kg)	51	79	69	100	66	80	83

Examining the data you can see that, as a rule, weight *does* increase with height, though there are some exceptions.

The Pearson coefficient (of a collection of samples with two features, like height and weight above) is always between -1 and 1. A Pearson coefficient of exactly 1 (or exactly -1) indicates a perfectly linear relationship between the two quantities, whereas a coefficient near zero indicates that there is no such nice relationship. The data above has a Pearson coefficient of $\approx .98$, indicating an extremely strong relationship between the variables.

Given two lists, $X = (x_1\ x_2\ \ldots\ x_n)$ and $Y = (y_1\ y_2\ \ldots\ y_n)$, each containing n values, Pearson's Coefficient is defined by the expression:

$$\frac{\sum_{i=1}^{n}(x_i - \bar{X})(y_i - \bar{Y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{X})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{Y})^2}},$$

where $\bar{X}$ and $\bar{Y}$ denote the averages of the x_i and the y_i, which is to say that

$$\bar{X} = \frac{1}{n}\sum_i x_i \quad \text{and} \quad \bar{Y} = \frac{1}{n}\sum_i y_i\,.$$

(a) Write a function, `list-sum`, that takes a list and returns its sum.

(b) Write a function that takes a list $X = (x_1\ x_2\ \ldots\ x_n)$ and returns the average $\bar{X}$. (Note: You will have to compute the length of the list in order to compute the average.) Call your function `average`.

(c) Write a function, `var-map`, that *maps* a list X to the "square of its deviation." Thus the list $X = (x_1\ x_2\ \ldots\ x_n)$ should be carried to the list

$$((x_1 - \bar{X})^2\ \ \ldots\ \ (x_n - \bar{X})^2)\,.$$

You should use the `map` function. (For example, your function, when evaluated on the list (1 2 3 4 5), should return (4 1 0 1 4).)

(d) Write a function `stdev` that returns the *standard deviation* of a list. Specifically, given the list $(x_1 \ldots x_n)$, your functions should return

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{X})^2}\,.$$

This should be easy, you already have the functions you need.

(e) Write a new version (called `map2`) of the `map` function which operates on two lists. Your function should accept three parameters, a function f and two lists X and Y, and return a new list composed of the function f applied to corresponding elements of X and Y. For concreteness, place the function first among the parameters, so that a call to the function appears as `(map2 f X Y)`. Your function may assume that the two lists have the same length. In particular, given two lists $(x_1\ x_2\ \ldots\ x_n)$ and $(y_1\ y_2\ \ldots\ y_n)$ (and the function f), `map2` should return

$$(f(x_1, y_1)\ \ldots\ f(x_n, y_n))\,.$$

(f) Write a function, `covar-elements`, that, given two lists $X = (x_1\ x_2\ \ldots\ x_n)$ and $Y = (y_1\ y_2\ \ldots\ y_n)$, returns the covariance list:

$$((x_1 - \bar{X})(y_1 - \bar{Y})\ \ \ldots\ \ (X_n - \bar{X})(Y_n - \bar{Y}))\,.$$

Note that the resulting list should have the same length as the two input lists; the ith element of the resulting list is the product $(x_i - \overline{X})(y_i - \overline{Y})$.

(g) Write a function, `pearson`, that computes Pearson's Coefficient. It might be useful to observe that an equivalent way to write Pearson's coefficient, by dividing the top and bottom by n, is

$$\frac{{}^1\!/_n \sum_{i=1}^{n}(x_i - \bar{X})(y_i - \bar{Y})}{\sigma(X)\sigma(Y)}\,.$$

(Thus, your function should take two lists as arguments, and return the value indicated above.)

2. *Least-Squares Line Fitting.* Line fitting refers to the process of finding the "best" fitting line for a set of points. This is one of the most fundamental tools that natural scientists use to fit mathematical models to experimental data.

As an example, consider the red points in the scatter plot of Figure 1. They do not lie on a line, but there is a line that "fits" them very well, shown in black. This line has been chosen—among all possible lines—to be the one that minimizes the sum of *squares* of the vertical distances from the points to the line. (This may seem like an odd thing to minimize, but it turns out that there are several reasons that it is the "right" thing minimize.)

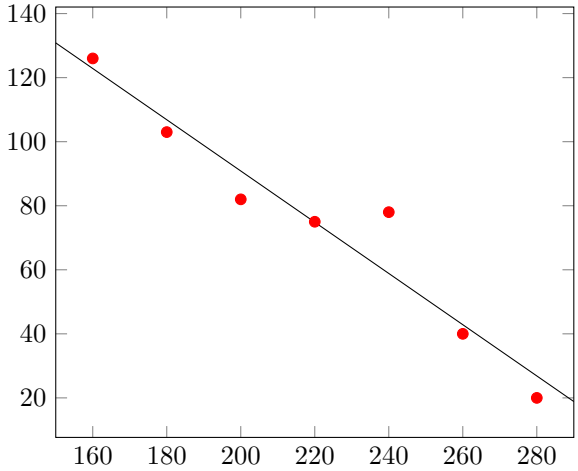


Figure 1: An example of a scatter plot and a best-fit line.

Since the equation of a line is $y = ax + b$, this boils down to finding a slope a and x-intercept b for given lists X and Y corresponding to point data. It turns out that the best fitting line can be found

with the following two equations:

$$a = r \cdot \frac{\sigma(Y)}{\sigma(X)}\,, \quad \text{and} \quad b = \bar{Y} - a\bar{X}\,,$$

where $\sigma(X)$ and $\sigma(Y)$ refer to the standard deviations of X and Y and r is Pearson's Coefficient.

(a) Write a function `best-fit` that takes two lists X and Y (of the same length) and returns a pair (a, b) corresponding to the (slope and intercept of the) best fit line for the data.

(b) Write an alternate version of `best-fit` that returns the best fitting line as a *function*. Specifically, write a new function `best-fit-fn` so that `(best-fit-fn pX pY)` returns the *function* which, given a number x, returns $ax + b$ where a and b are the best fit coefficients.

Use this to the best fit line for the following data:

$$X = \{160, 180, 200, 220, 240, 260, 280\}\,, \qquad Y = \{126, 103, 82, 75, 78, 40, 20\}\,.$$

Call the function `fitline` so that you can graph it in the next problem.

3. Define a Scheme function `(insert x l)` which takes a value x (an integer) and a *sorted* list ℓ (in increasing order) and inserts x at the right location within ℓ so that the list remains sorted. Note that `insert` produces a new list ℓ' identical to ℓ except for the addition of x at the right spot. For instance, the call

```
(insert 5 (list 1 2 4 6 7))
```

produces the list

```
(1 2 4 5 6 7)
```

As before, this leaves the input list ℓ in pristine condition.

4. Armed with `insert`, you are now ready to implement another sorting algorithm knows as *insertion sort*. The idea of the algorithm is straightforward. Given an unsorted list ℓ, it proceeds by peeling off elements from the front of ℓ and inserting them (one at a time of course) at their "sweet spot" within a sorted list ℓ' that starts off as an empty list. For instance the call

```
(insSort (list 3 5 1 6 9 0 2 7))
```

produces the list

```
(0 1 2 3 5 6 7 9)
```

Unsurprisingly, *insertion sort* is closely related to *selection sort* which we covered in class. Write a Scheme function `(insSort l)` which, given a list ℓ of integers, produces a sorted permutation of ℓ (in increasing order).

5. Define a Scheme function, `merge`, which takes two lists ℓ_1 and ℓ_2 as arguments. Assuming that each of ℓ_1 and ℓ_2 are *sorted* lists of integers (in increasing order, say), `merge` must return the sorted list containing all elements of ℓ_1 and ℓ_2.

To carry out the merge, observe that since ℓ_1 and ℓ_2 are already sorted, it is easy to find the smallest element among all those in ℓ_1 and ℓ_2: it is simply the smaller of the first elements of ℓ_1 and ℓ_2. Removing this smallest element from whichever of the two lists it came from, we can recurse on the resulting two lists (which are still sorted), and place this smallest element at the beginning of the result.

6. You guessed it! There is a third sorting algorithm that one can build from the Scheme function `merge`. It is aptly named *merge sort*. Its architecture is based on the simple principle known as "divide and conquer" (like `quickSort`, also covered in class). It works as follows: given a list ℓ, split the list into two sub-lists ℓ_1 and ℓ_2 of approximately the same length (a difference of 1 at most) such that all elements of ℓ appear in either ℓ_1 or ℓ_2. For instance, a list $\ell =$`(1 6 7 3 9 0 2)` could be split into $\ell_1 =$`(1 7 9 2)` and $\ell_2 =$`(6 3 0)`. Then one can recursively sort ℓ_1 and ℓ_2 to obtain sorted versions ℓ_1' and ℓ_2' and *merge* them to recover a fully sorted list. Write a Scheme function `(mergeSort l)` which, given an unsorted list ℓ of integers, returns a sorted version of ℓ's content. **Hint:** writing a helper function to carry out the splitting would be a wise first step. Hiding that helper function in the bowels of `mergeSort` would be even better! A whimsical illustration of mergeSort is shown in Figure 2.

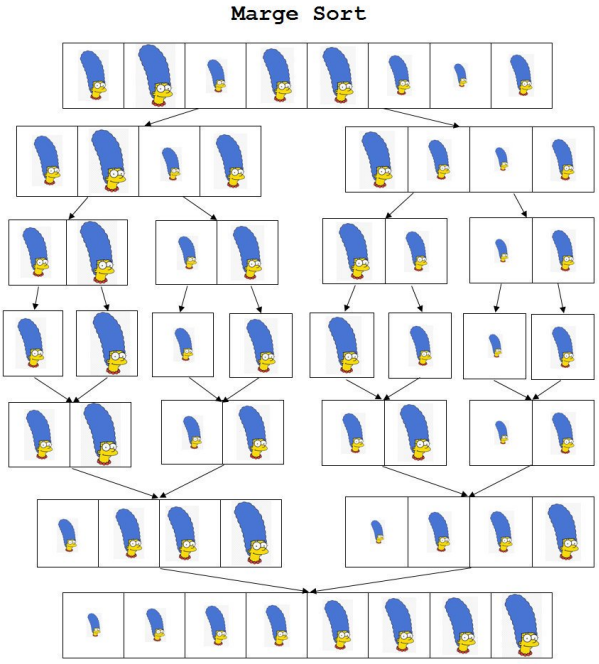


Figure 2: `mergeSort` at work!

7. Given a list $\ell = (x_1\, x_2\, \ldots\, x_{n-1}\, x_n)$, we define two *rotations* of the list: The *left-rotation* is $(x_2\, \ldots\, x_{n-1}\, x_n\, x_1)$ and, likewise, the *right-rotation* is $(x_n\, x_1\, x_2\, \ldots\, x_{n-1})$. (These rotation operations do not change the empty list, or any list of length 1.) Define the functions `rotate-right` and `rotate-left` to carry out these operations on a list. Once your functions have been correctly defined, you should find that

```
> (rotate-right '(1 2 3))
'(3 1 2)
> (rotate-right '(1))
'(1)
> (rotate-left '(1 2 3))
'(2 3 1)
> (rotate-left '(1))
'(1)
> (rotate-right (rotate-left '(1 2 3 4 5)))
'(1 2 3 4 5)
```

CSE1729 – Introduction to Programming | October 25, 2019

Problem Set 7

Recall the conventions we have adopted in class for maintaining trees. We represent the empty tree with the empty list `()`; a nonempty tree is represented as a list of three objects

```
(value left-subtree right-subtree)
```

where `value` is the value stored at the root of the tree, and `left-subtree` and `right-subtree` are the two subtrees. We introduced some standardized functions for maintaining and accessing this structure, which we encourage you to use in your solutions below.

```
(define (make-tree value left right) (list value left right))
(define (value tree) (car tree))
(define (left  tree) (cadr tree))
(define (right tree) (caddr tree))
```

1. *SICP Exercise 2.31.* Define a SCHEME procedure, named `tree-map`, which takes two parameters, a tree, `T`, and a function, `f`, and is analogous to the `map` function for lists. Namely, it returns a new tree T' with a topology identical to that of `T` but where each node $n \in T'$ contains the image under f of the value stored in the corresponding node in T. For instance, if the input tree is shown in Example 1 and the function `f` is $f(x) = x^2$. then the tree returned by `tree-map` is shown in Example 2.

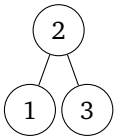


Figure 1: `T`, the tree passed to `tree-map`

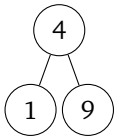


Figure 2: T', the tree returned by `tree-map` when passed `T` and the squaring function as parameters.

2. Define a SCHEME procedure, named `tree-equal?`, which takes two trees as parameters and returns `#t` if the trees are identical (same values in the same places with exactly the same structure) and `#f` otherwise. Use the SCHEME `eq?` function to test equality for the values of the nodes.

3. Define a SCHEME procedure, named `(tree-sort l)`, which takes a list of numbers and outputs the same list, but in sorted order. Your procedure should sort the list by

   (a) inserting the numbers into a binary search tree and, then,

   (b) extracting from the binary search tree a list of the elements in sorted order.

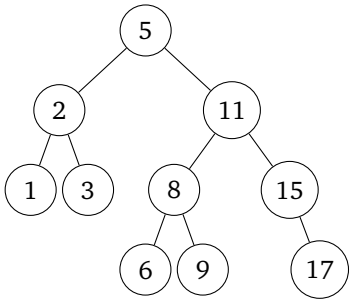
   To get started, write a procedure called `(insert-list L T)` which takes a list of numbers L and a binary search tree T, and returns the tree that results by inserting all numbers from L into T. (Place the argument L first, so a call to your function should have the form `(insert-list L T)`, where L is a list and T is a (perhaps empty) binary search tree.)

   Then write a function called `sort-extract` which takes a binary search tree and outputs the elements of the tree in sorted order. (We did this in class!)

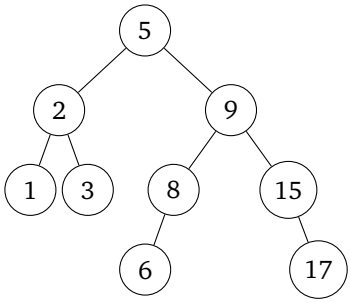
   Finally, put these two functions together to achieve `(tree-sort l)`. (Note, all three of these functions will be graded, so your solutions must consist of three top-level functions, `insert-list`, `sort-extract`, and `tree-sort`.)

4. Define a SCHEME procedure, named `(delete-value T v)`, that takes a binary search tree, `T`, and a value, `v`, as arguments and returns the binary search tree that results from removing the node containing the value v. (The tree T should be the first argument.) You may assume that the tree contains no more than one node with value v, but your procedure should be well-behaved if called with a tree T that does not contain the value v: in this case, it should return the tree unchanged. Note that the tree your function returns *must maintain the binary search tree property*. Removing interior (that is, non-leaf) nodes requires a little care. (See Example 3 for an example.) Details are given below.

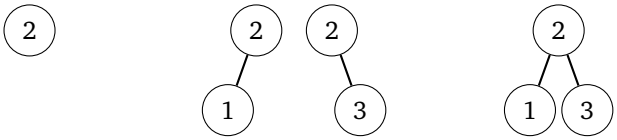
   Deleting nodes from trees The `delete-value` function must remove the one occurrence of v from the input binary search tree T. Naturally, if T has n nodes, the output tree T' has $n-1$ nodes; note that T' must satisfy the binary search tree property (*for every node w, the nodes in the left sub-tree of w hold values smaller than the value held in w and the nodes in the right-subtree of w hold values larger than the value held in w*). There are number of cases that need to be handled separately: the node n may have no sub-trees, exactly one subtree, or two subtrees. (See Example 5.) The first two situations can be handled easily; just remove the node and "replace" it with its single subtree. The third case, illustrated below in Example 4, requires that you restructure the tree a bit to reattach the two "orphaned" subtrees of n in an appropriate way. There are two natural choices: either replace the value at the deleted node with the largest value in the left subtree or the smallest value in the right subtree while removing the corresponding leaf node. In your implementation, please adopt the first of these conventions: specifically, in order to remove a node n with two subtrees, replace the value of n with the *largest* value in the left subtree, and then remove this value from the left subtree. (See the example below.)



Example 3: An example of a binary search tree.



Example 4: The same binary search tree after the removal of the value 11. Note, we chose to promote the largest value in the left subtree (9) to the node vacated by the value 11.



Example 5: Three possibilities when finding a node to remove. (The middle two cases are treated as one.)

5. This problem concerns ways to represent arithmetic expressions using trees. For this purpose, we will consider 4 arithmetic operations: $+$ and $*$, both of which take two arguments, and $-$ and $1/\square$, both of which take one argument. (As an example of how these one-argument operators work, the result of applying the operator $-$ to the number 5 is the number -5; likewise, the result of applying the $1/\square$ operator to the number 5 is the number $1/5$.)

An *arithmetic parse tree* is a special tree in which every node has zero, one, or two children and:

- each leaf contains a numeric value, and
- every internal node with exactly two children contains one of the two arithmetic operators $+$ or $*$,
- every internal node with exactly one child contains one of the two arithmetic operators $-$ or $1/\square$.

(You may assume, for this problem and the next, that when a node has a single child, this child appears as the left subtree.)

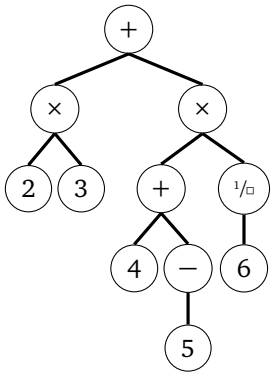
If T is an arithmetic parse tree, we associate a value with T (which we call value(T)) by the following recursive rule:

- if T has a single (leaf) node, value(T) is equal to the numeric value of the node,
- if T has two subtrees L and R, and its root is the operator $+$, then value(T) = value(L) + value(R),
- if T has two subtrees L and R, and its root is the operator $*$, then value(T) = value(L) $*$ value(R),
- if T has one subtree, S, and its root is the operator $-$, then value(T) = $-$value(S),
- if T has one subtree, S, and its root is the operator $1/\square$, then value(T) = $1/$ value(S).

You can see how to associate with any arithmetic expression a natural arithmetic parse tree. Note that since $-$ and $1/\square$ are *unary* operators (take only one argument), you have to give a little thought to how to represent expressions such as $3-5$ or $3/5$. For example, the arithmetic parse tree for the expression

$$2 \times 3 + \frac{4-5}{6}$$

is shown in Example 6.



Example 6: An arithmetic parse tree for the expression $2 \times 3 + (4 + (-5)) \times (1/6)$.

Write a SCHEME function `(arithvalue T)` which, given an arithmetic parse tree T, computes the value associated with the tree. You may assume that the operators $+$, $\times$, $-$, and $1/\square$ appear in the tree as the *characters* `#\+`, `#\*`, `#\-`, and `#\/`. (See the comments at the end of the problem set concerning the SCHEME character type.) To test your code, try it out on the parse tree

```
(define example (list #\+ (list #\*
                                (list 4 '() '())
                                (list 5 '() '()))
                      (list #\+
                            (list #\/ (list 6 '() '()) '())
                            (list 7 '() '()))))
```

6. (A continuation of the previous problem; pre-, in-, and post-order traversal.) Such (arithmetic) trees can be traversed recursively (which you will have done in the solution to your previous problem). In this problem, you will print out the expression associated with a parse tree, using several different conventions for writing arithmetic expressions.

   There are three conventional orders in which nodes and subtrees can be "visited" during a recursive traversal of a tree. Note that at each node, there three tasks to carry out: visit (in this case, that means *print out*) the node, traverse the left subtree, and traverse the right subtree. For instance, an *inorder* traversal of a binary tree will:

   1) recursively traverse the left subtree,
   2) visit the node, and then
   3) recursively traverse the right subtree.

   Therefore, an inorder traversal yields *infix notation*. Likewise, a preorder traversal visits each node first and then recursively traverses the left and then the right subtrees. A preorder traversal produces the expression in *prefix notation* as shown in Example 7. And, yes, as you may have guessed, a postorder traversal examines the left subtree, then the right subtree and finally visits the root node itself. A postorder traversal produces the same expression in postfix notation. The postfix notation for the example expression is "$23\times45-+6\frac{1}{\square}\times+$".

   (a) Define a SCHEME function `arith-prefix` that takes an arithmetic expression tree and uses a preorder scan on the tree to produce the expression in prefix (also called "Polish") notation. Your function should return a *string* containing the expression in prefix notation. See the following section regarding characters and strings in SCHEME.

   One difficulty is that your tree has values of different "types"—the leaves have numbers, the internal nodes have characters. You will need to convert everything to strings. You could do this by hand, or you can map the following function on your tree (using your previous solution to `tree-map`).

```
(define (prepare x)
  (cond ((number? x) (number->string x))
        ((char? x) (string x))))
```

   This function will convert all characters and numbers to strings, so that your tree contains only string labels. Once the tree contains only strings, you can build the final expression by appending the strings together to build your final expression.

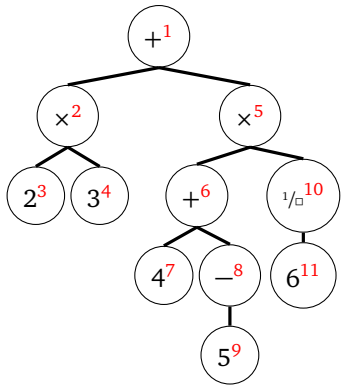
   (Note that the function `number?` returns true if its argument is a number; likewise, the function `char?` returns true if its argument is a character. The function `number->string` converts a number to a string.)

   As an example, applied to the example tree of Figure 6 above, your function `arith-prefix` should produce the string output `"+*23*+4-5/6"`.

   (b) Define a SCHEME function `arith-postfix` that takes an arithmetic expression tree as a parameter and uses a postorder traversal to produce a string representing the expression in postfix notation (also called "Reverse Polish Notation," or RPN). After the previous problem, this is easy! Incidentally, several early computer and calculator architectures were designed with RPN. Several current languages such as PostScript and software programs including the MacOSX calculator, the Unix dc calculator program and several Android and iPhone apps make use of RPN.

   As an example, applied to the example tree of Figure 6 above, your function `arith-postfix` should produce the string output `"23*45-+6/*+"`.

   (c) It is an interesting fact that when the numbers are only a single digit long, the postfix expression associated with an arithmetic tree *completely determines the tree*; that same is true for the prefix expression.



Example 7: An arithmetic parse tree showing "preorder" numbering of the nodes (in red). This gives the expression in prefix notation as "$+ \times 23 \times +4 - 5\frac{1}{\square}6$."

> This is not true for infix expressions: "2+3*6" can be generated from two different arithmetic trees (which given different values!). Define a SCHEME function, named `arith-infix`, which converts a tree into an infix expression but adds parentheses around the arguments of every $+$, $\times$, $-$, and $\frac{1}{\square}$. To be more precise, a $\times$ or $+$ operator produces output of the form $(a_1 \times a_2)$ (or $(a_1 + a_2)$), whereas the operators $-$ and $\frac{1}{\square}$ produce output of the form $-(a_1)$ (or $/(a_1)$). Thus, your algorithm, when applied to the tree pictured above, should yield the string `"((2*3)+((4+-(5))*/(6)))"`. Note that the parentheses do uniquely determine the tree from which the expression arose.

Strings and characters in SCHEME SCHEME has the facility to work with strings and characters (a *string* is just a sequence of characters). In particular, SCHEME treats *characters* as atomic objects that evaluate to themselves. They are denoted: `#\a`, `#\b`, ... Thus, for example,

```
> #\a
#\a
> #\A
#\A
> (eq? #\A #\a)
#f
> (eq? #\a #\a)
#t
```

The "space" character is denoted `#\space`. A "newline" (or carriage return) is denoted `#\newline`.

A *string* in SCHEME is a sequence of characters, but the exact relationship between strings and characters requires an explanation. A string is denoted, for example, `"Hello!"`. You can build a string from characters by using the `string` command as shown below. An alternate method is to use the `list->string` command, which constructs a string from a list of characters, also modeled below. Likewise, you can "explode" a string into a list of characters by the command `string->list`:

```
> (string #\S #\c #\h #\e #\m #\e)
"Scheme"
> (list->string '(#\S #\c #\h #\e #\m #\e))
"Scheme"
> (string->list "Scheme")
(#\S #\c #\h #\e #\m #\e)
```

```
> "Scheme"
"Scheme"
```

Finally, given two strings, you can append them together using the `string-append` function (alternatively, you could turn them in to lists, append those, and convert back):

```
> (string-append "book" "worm")
"bookworm"
> (list->string (append (string->list "book") (string->list "worm")))
"bookworm"
```

Note that strings, like characters, numbers, and Boolean values, evaluate to themselves.

CSE1729 – Introduction to Programming | October 31, 2019

Problem Set 8

Recall the discussion from class on Huffman trees. In particular, to construct an optimal encoding tree for a family of symbols $\sigma_1, \ldots, \sigma_k$ with frequencies $f_1, \ldots, f_k$, carry out the following algorithm:

1. Place each symbol σ_i into its own tree; define the weight of this tree to be f_i.
2. If all symbols are in a single tree, return this tree (which yields the optimal code).
3. Otherwise, find the two current trees of minimal weight (breaking ties arbitrarily) and combine them into a new tree by introducing a new root node, and assigning the two light trees to be its subtrees. The weight of this new tree is defined to be the sum of the weights of the subtrees. Return to step 2.

Strings and characters in SCHEME SCHEME has the facility to work with strings and characters (a *string* is just a sequence of characters). In particular, SCHEME treats *characters* as atomic objects that evaluate to themselves. They are denoted: `#\a`, `#\b`, ... Thus, for example,

```
> #\a
#\a
> #\A
#\A
> (eq? #\A #\a)
#f
> (eq? #\a #\a)
#t
```

The "space" character is denoted `#\space`. A "newline" (or carriage return) is denoted `#\newline`. A *string* in SCHEME is a sequence of characters, but the exact relationship between strings and characters requires an explanation. A string is denoted, for example, `"Hello!"`. You can build a string from characters by using the `string` command as shown below. An alternate method is to use the `list->string` command, which constructs a string from a list of characters, also modeled below. Likewise, you can "explode" a string into a list of characters by the command `string->list`:

```
> (string #\S #\c #\h #\e #\m #\e)
"Scheme"
> (list->string '(#\S #\c #\h #\e #\m #\e))
"Scheme"
> (string->list "Scheme")
(#\S #\c #\h #\e #\m #\e)
> "Scheme"
"Scheme"
```

Note that strings, like characters, numbers, and Boolean values, evaluate to themselves.

1. Write a SCHEME function `(get-count text)` which, given a string `text` computes a list of character counts appearing in the text. Namely, the call `(get-count "hello")` returns the list `((#\o . 1) (#\l . 2) (#\e . 1) (#\h . 1))`.

2. Write a SCHEME function which, given a list of character counts, returns a list of character frequencies. Namely, a call `(get-freq (get-count "hello"))` returns the list `((#\o . 1/5) (#\l . 2/5) (#\e . 1/5) (#\h . 1/5))`.

3. Write a SCHEME function `(huffman freqs)` which, given a list of characters and frequencies, constructs and returns a Huffman encoding tree. You may assume that the characters and their frequencies are given in a list of pairs: for example, the list

```
((#\a . 1/5) (#\b . 1/7) (#\c . 1/9) (#\d . 172/315))
```

represents the 4 characters a, b, c, and d, with frequencies 1/5, 1/7, 1/9, and 172/315, respectively. Given such a list, you wish to compute the tree that results from the above algorithm. I suggest that you maintain nodes of the tree as lists: internal nodes can have the form

```
('internal weight 0-tree 1-tree)
```

where `internal` is a token that indicates that this is an internal node, and `0-tree` and `1-tree` are the two subtrees; leaf nodes can have the form

```
('leaf weight symbol)
```

where `symbol` is the character held by the leaf. Note that you will have to use the SCHEME `quote` command to construct both types of nodes: for example, to construct an internal node with the two subtrees `0-tree` and `1-tree`, you could use the procedures below

```
(define (htree-leaf letter weight) (list 'leaf weight letter))
(define (htree-node t0 t1) (list 'internal (+ (htree-weight t0)
                                              (htree-weight t1)) t0 t1))
(define (htree-weight t) (cadr t))
```

Observe how the weight of an internal node is simply the sum of the weights of the sub-trees. For convenience, you will also see above a helper function `(htree-weight t)` meant to return the weight of a tree. Note how the weight is always the second value in the list describing a tree node (whether internal or a leave). When you traverse a Huffman coding tree, you can determine if a given node is an internal node by deciding if the `car` of the list associated with that node is the token `internal` (Similarly, you can check if a node is a leaf). The process for building a Huffman tree simply processes a work queue of weighted trees selecting the two *lightest* trees (smallest weights), combining them and placing the composite back into the work queue. The process ends when the work queue has a single tree (and that tree is the result of your function!). (**Hint:** since we are interested in light trees, a heap might come in handy!).

4. Define a SCHEME function `(codeWords t)` that takes, as input, a Huffman coding tree t and outputs a list of pairs $\langle symbol, code\$rangle$ containing the elements at the leaves of the tree ($symbol$) along with their associated encodings as a string over the characters `#\0` and `#\1` (the $code$). For example, given the tree of Figure 1, your function should return the list

```
((#\a . "0") (#\b . "10") (#\c . "10")) .
```

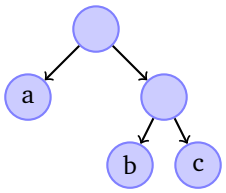


Figure 1: A Huffman tree yielding an encoding of the three symbols a, b, and c.

CSE1729 - Introduction to Programming | November 15, 2019

Problem Set 9

Practice with SCHEME objects

1. **Encapsulation** - protecting object state (data) behind an abstraction barrier.
   Define a SCHEME object, named `make-rat`, which creates an object representing a rational number. Recall that when we introduced pairs, we discussed a representation of rationals using a SCHEME pair. Here, your rational numbers will be represented with *objects*. Your rational object should expose the following capabilities for a rational number: get numerator, get denominator, get reciprocal, add, subtract, and multiply. Thus, your object should expose the following methods in its dispatcher:

```
(define (make-rat a b)
        ...
    (lambda (method)
      (cond ((eq? method 'numer) get-numer)
            ((eq? method 'denom) get-denom)
            ((eq? method 'reciprocal) reciprocal)
            ((eq? method 'add) add)
            ((eq? method 'sub) sub)
            ((eq? method 'mult) mult)))))
```

   Note, your rational number object should represent rationals in reduced form. You may find SCHEME *builtin function* `(gcd a b)`*, which computes the greatest common divisor of its two arguments* `a` *and* `b`*, useful.*

2. Define a SCHEME object—call it `basic-set`—that maintains an (initially empty) set of numbers. Your object should have methods that correspond to

   - `empty`, which should return `#t` if the set is empty, and `#f` otherwise;
   - `insert`, which should insert a new element into the set; and
   - `delete`, which should delete a particular element from the set; and
   - `element?`, which should determine if a given element is in the set.

   Implement the set as a standard SCHEME list, and return a dispatcher that provides access to your methods, as we have done in class, via a token.

   Thus, your object should behave as follows:

```
> (define newset (basic-set))
> ((newset 'empty))
#t
> ((newset 'insert) 2)
> ((newset 'element?) 2)
#t
> ((newset 'delete) 2)
> ((newset 'empty))
#t
```

3. Define a SCHEME object—call it `stat-set`—that maintains an (initially empty) set of numbers along with several statistics pertaining to the set. Your object should have methods that correspond to

- empty, which should return `#t` if the set is empty, and `#f` otherwise;
- insert, which should insert a new element into the set;
- element?, which should determine if a given element is in the set;
- largest, which should return the largest element in the set;
- smallest, which should return the smallest element in the set;
- size, which should return the size of the set; and
- average, which should return the average of the elements in the set.

Note that your set does *not* need to implement delete. Implement the set as a standard SCHEME list, and return a dispatcher that provides access to your methods, as we have done in class, via a token.

Important amplification. Your `stat-set` methods that implement `largest`, `smallest`, `size`, and `average`, should use only a *constant* amount of computation to determine their return value: specifically, the amount of time they take to return should not depend on the size of the set. (Thus you *cannot* scan the list to determine its length, or the maximum element of the list.)

How is this possible? One strategy is to set up some "private variables" in your object called, e.g. `current-max`, `current-min`, `current-length`, etc. These variables should maintain, at all times, the current values of these quantities. When a new number is inserted, you will have to update your list (that maintains the set) and, in addition, update each of these statistics appropriately.

I think that the easiest way to "maintain" the current average is to actually maintain the current *sum* of all the elements in the set, and—when the user asks for the average—return the sum divided by the current size.

Incidentally, why didn't we ask you to implement the delete operation?

4. Using the conventional set notation $\{1,2,2\}$ and $\{1,2\}$ denote the same set (which contains the numbers 1 and 2 and no others): specifically, as far as a set is concerned, an element either appears or it does not—there is no notion of an object appearing more than once.

   A *multiset* is like a set in which elements can appear more than once. People often use the same "set style" notation for multisets: e.g., $\{1,2,2\}$ denotes the multiset in which 1 appears once and 2 appears twice. Of course, as multisets $\{1,2,2\}$ and $\{1,2\}$ are different.

   Notice that to maintain a multiset (of numbers, say), one needs to keep track of *how many times* each number appears in the set (which could be zero).

   Consider the representation of a multiset as a sorted list of pairs, where the first part of each pair represents an integer, and the second represents the number of times it appears (a non-negative integer). Thus the multiset $\{1,2,2,3,8,8\}$ would be represented as the list `((1 . 1) (2 . 2) (3 . 1) (8 . 2))`. (Note that numbers which do not appear at all in the multiset do not have a pair in the list.)

   Define a multiset object, which provides the methods:

   - empty, which should return `\#t` if the multiset is empty, and `\#f` otherwise;
   - insert, which should insert a new element into the set; of course, if the element already exists in the multiset, the *number of times* it appears in the set should increase;
   - multiplicity?, which should return the number of times a given element appears in the multiset (it should return 0 if the element does not appear at all);
   - delete, which should delete one appearance of a particular element from a multiset; and
   - delete-all, which remove all appearances of an element in a multiset.

Stack Applications

5. In this problem you will define an object that implements the *Stack* abstract data type *using a list to store the elements*. When the object is created, it starts as an empty stack.

   To implement the stack, the object will initially create an empty list, which it will store the elements of the stack. The stack contents are then maintained with the following convention: to represent the stack containing the elements $e_1, e_2, \ldots, e_n$ (where e_1 is the *top* of the stack and e_n is the *bottom* of the stack) the list will contain the elements as shown in Figure 1, just below. Observe that the bottom element of the stack

   $$'(e_1\ e_2\ e_3 \ldots e_k\ )$$

   Figure 1: Layout of a stack in a list.

   is always at the end of the list. The top element of the stack can be accessed at the front of the list. To place a new element on the top of the stack, one needs only add it to the front of the list. Popping an element off the top of the stack is handled by returning the appropriate element *(take particular care to ensure you are returning the proper value when using destructive assignment)* and "removing" that element from the front of the list. Your object should expose methods for

   empty? Returns a Boolean value (`\#t` or `\#f`) depending on whether the stack is empty.

   push Pushes a new element onto the top of the stack.

   pop Pops off the top element from the stack and returns it.

   top Returns the value of the top of the stack (without changing the contents of the stack).

   Thus, your object should have the form:

```
(define (make-stack)
  (let (...)                              ;; internal stack variables
    (define (empty?) ...)          ;; stack methods
    (define (push x) ...)
    (define (pop) ...)
    (define (top) ...)
    (define (dispatcher ...) ...) ;;the dispatcher
    dispatcher))
```

6. (**Evaluation of Postfix Expressions**) Given a list of operands and operators that represent a postfix expression, one can use a stack to evaluate the postfix expression. Its true! The algorithm to do this is as follows:

Algorithm 1 Evaluate Postfix Expression

```
repeat
  if operand at front of input string then
    push operand onto stack
  else
    pop stack to remove operands
    apply operator to operand(s)
    push result onto stack
  end if
until input string is empty
```

For example, to evaluate the expression "23 15 +" we would:

(a) Push 23 onto the stack

(b) Push 15 onto the stack

(c) "+" is an operator, so

  i. Pop second operand *(note the operands are popped in reverse order)*

  ii. Pop first operand

  iii. Apply the operator (in this case, addition)

  iv. Push the result back onto the stack

(d) Once the expression has been evaluated, the result is on the top of the stack.

Define a SCHEME function, named `(eval-postfix p)`, that will take a postfix expression (stored in a list of integers representing operands and characters representing operators), evaluate that expression, and return the result.
Your function should support the operations of addition (`#\+`), subtraction (`#\-`), multiplication (`#\*`), division(`#\/`), and exponentiation(`#\^`).
You may want to work incrementally by starting with a function that takes a character representing an operator and can pop two operands off of a stack, evaluate the operator and push the result back on the stack. Next you can add a function that evaluates a postfix expression by pushing operands onto the stack and evaluating operators when encountered. Note, you may want to use the `number?` function to determine whether an item in the list is an operand or an operator.

CSE1729	Introduction to Programming Problem Set 10	December 1, 2019

STREAM REFRESHER

Recall[1] that a SCHEME stream is a pair of the form

```
(a . (lambda () ...)).
```

The `car` of the pair is the first element of the stream. The job of the `cdr` of the pair (the lambda expression) is to return the rest of the stream (which is another stream, and must have exactly the same structure) when it is called with no arguments. As you recall, SCHEME provides two keywords `delay` and `force` that automate the process of creating a delaying lambda (the `cdr` of a pair) and triggering its execution. Namely, the following

```
(define s (cons 1 (delay (...))))
(force (cdr s))
```

is equivalent to

```
(define s (cons 1 (lambda () (...))))
((cdr s))
```

i.e., `delay` creates a lambda while `force` forces its execution. In this assignment you will make use of `delay` and `force` to implement a whole array of streams.

Using the stream representation convention (a pair), one can define

```
(define (head stream)  (car stream))
(define (rest stream)  (force (cdr stream)))
```

As an example, the function `integer-stream` returns the stream of positive integers $a, a+1, a+2, \ldots$:

```
(define (integer-stream a)  (cons a (delay (integer-stream (+ 1 a)))))
```

To wrap up our example

```
> (define a (integer-stream 5))
> a
'(5 . #<promise>)
> (rest a)
'(6 . #<promise>)
> (rest (rest a))
'(7 . #<promise>)
> (head (rest (rest a)))
7
```

Where a `promise` is just the lambda waiting to be *forced*.

[1] The material will be covered during the two upcoming lectures!

There are natural ways to *operate* on streams. For example, the following function *shifts* a stream of integer by a constant

```
(define (shift s c)
   (cons (+ (head s) c)
         (delay (shift (rest s) c))))
```

Then, for example

```
> (define a (shift (integer-stream 0) 42))
> a
'(42 . #<procedure>)
> (rest a)
'(43 . #<procedure>)
> (rest (rest a))
'(44 . #<procedure>)
```

You will notice that in the example above, we created a function `shift` that returned the stream; when such a stream is created, it will have embedded in the `cdr` of the stream a call to this function. We mention that there are circumstances where it may be helpful for the definition of the stream to carry additional information (beyond simply the first element of the stream). As an example, consider the stream of Fibonacci numbers.

```
(define (fib-stream current next)
  (cons current (delay (fib-stream next (+ current next)))))
```

To generate the stream of regular Fibonacci numbers, you would use the call `(fib-stream 0 1)`.

Please use the stream conventions above. **Do not use lambdas directly as we cannot mix both conventions and this will break your submission when graded. Everything should be done in term of `delay` and `force`.**

In all of the following, you can assume the existence of a stream named `nat` and defined as follows:

```
(define (integer-stream x)
    (cons x  (delay (integer-stream (+ 1 x)))))
(define nat (integer-stream 0))
```

---

1. Write a function `(take s k)` which, given a stream s and an integer k extracts from s the list of the first k elements. For instance, the call `(take nat 5)` produces `(0 1 2 3 4)`

2. Write a function `(square s)` which, given a stream s computes a new stream consisting of the squares of all the values in s. For instance, the call `(take (square nat) 5)` produces `(0 1 4 9 16)`

3. Write a function `(cube s)` which, given a stream s computes a new stream consisting of the cubes of all the values in s. For instance, the call `(take (cube nat) 5)` produces `(0 1 8 27 64)`

4. Write a function `(merge-streams s t)` that *merges* two streams. Specifically, if s and t are two streams of positive numbers in increasing order, `(merge-streams s t)` should be the stream containing all elements appearing in either s or t, also in increasing (sorted) order. For example, if s is the stream of all perfect squares

   $$0, 1, 4, 9, 16, 25, 36, \ldots$$

   and t is the stream of all perfect cubes

   $$0, 1, 8, 27, 64, 125, 216, \ldots$$

then `(merge-streams s t)` would be the stream

$$0, 0, 1, 1, 4, 8, 9, 16, 25, 27, 36, \ldots .$$

Note that there are numbers, such as 1 and $(2^3)^2 = (2^2)^3 = 2^6 = 64$ that are *both* perfect squares and perfect cubes. In this case, your merged stream should produce the number twice.

5. Write a function `(repeat x)` which, given a value x, produces a stream that repeats x ad infinitum. For instance, the call `(take (repeat 42) 5)` produces `(42 42 42 42 42)`

6. Write a function `(weave s1 s2)` which, given two streams s_1 and s_2 produces a new stream interlacing the two streams perfectly. Namely, the new stream alternates the values from s_1 and s_2. For instance, the call `(take (weave (rest nat) (repeat 0)) 10)` produces `(1 0 2 0 3 0 4 0 5 0)`.

7. Write a function `(mul s1 s2)` which, given two streams s_1 and s_2 produces a new stream of the point wise products of values from s_1 and s_2. Namely, given the statement

```
(take (mul nat (repeat 2)) 10)
```

   One would obtain the stream extract: `(0 2 4 6 8 10 12 14 16 18)`.

8. Write a function `(frac s)` which, given a stream s produces a new stream of the inverse of each value in s Namely, given the statement

```
(take (frac (rest nat)) 10)
```

   One would obtain the stream extract: `(1 1/2 1/3 1/4 1/5 1/6 1/7 1/8 1/9 1/10)`.

9. Write a SCHEME function `(pow x)` which, given a value x computes the stream of powers of x, i.e., it computes the infinite stream $x^0, x^1, x^2, x^3, \ldots$. Importantly, you **cannot** use the SCHEME function `expt`. Instead, you should define the power purely in term of streaming operations. There are several ways to achieve this. For instance, you can realize that the stream seeded at x^i is easily derived from the stream seeded at x^{i-1} as it suffices to multiply its seed by x to obtain $x^i = x^{i-1} \cdot x$. The `mul` and `repeat` streams may come in handy! The same identity $x^i = x^{i-1} \cdot x$ can also help you define an auxiliary function to obtain the next value in the stream and see it manually (all from first principles).

10. Write a SCHEME function `facts` (no arguments!) which produces the stream (0!1!2!3!4!5!..., i.e., it is the stream of factorial values. Interestingly, the idea to produce the stream is related to how you might have implemented the `pow` streamer. Namely, The i^{th} entry of the stream of factorials is *the result of the product of the* $(i-1)^{th}$ *value of the stream and the* i^{th} *entry of the stream of naturals (starting from 1)*. Indeed, $i! = (i-1)! \cdot i$.

11. **The Return of The Power Series**. As you surely recall, one can use a power series to approximate some classic functions. Indeed, we know that

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

   And one can write a classic function that approximates e^x by using a finite number of terms in the summation. Namely, if we have a list lst of k values representing the first k terms of this summation, then $e^x \approx$`(fold-right lst + 0)`. The more terms we have, the more precise the approximation is. If we only produce the *stream* t of summation terms, a call like `(define lst (take t 20))` would bind lst to the first 10 terms of the summation. Our task is clear! Write a SCHEME function `(expts x k)` which, given a value x and a number of terms k computes an approximation of e^x with k terms. The code should have this general shape (add the meaty part!)

```
(define (expts x k)
  (fold-right (take ...  k) + 0))
```

where the ellipsis is replaced by the required stream.