CSE1729 - Introduction to Programming September 24, 2018
Problem Set 4

1. Abstracting the product of a series
   (a) The lab assignment function `sum` is an abstraction of the sigma notation for summation:
   $$\sum_{n=a}^{b} f(n) = f(a) + \cdots + f(b)$$
   Write a function `(product term a next b)` that abstracts the pi notation for the product of a series. You should name your function `(product-i term a next b)` if you choose to write an iterative version of the function.
   $$\prod_{n=a}^{b} f(n) = f(a) \times \cdots \times f(b)$$
   (b) If your `product` function is in a recursive form, write a second one that is in the iterative form named `(product-i term a next b)`, and vice-versa (that is, you should write 2 versions of `product`, one recursive and one iterative while following the naming convention).
   (c) Use your product functions to define an function `(pi-approx n)` that returns an approximation of π using the first n terms of the following:
   $$\frac{\pi}{4} = \frac{2 \times 4 \times 4 \times 6 \times 6 \times 8 \cdots}{3 \times 3 \times 5 \times 5 \times 7 \times 7 \cdots}$$
   You will need to define appropriate functions for `term` and `next`.
   (d) Show that your pi-approx function works for 1, 100, and 1000. What do these tests demonstrate to you?

2. 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 so that `(der f h)` 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 h as arguments, and return a *function*.)

(b) Compare the derivative of $\sin(x)$, as computed by your function with h $= .5$, with the function $\cos(x)$ over at least 4 values $(0, \pi/2, \pi, 3\pi/4)$.

(c) Write a function `(fun x)` that computes $3x^2 - 2x + 7$. As with the previous, compare your calculated derivative with $6x - 2$ for a few values of x.

(d) Define a SCHEME function, named `(nth-deriv f n h)`, that takes three parameters, f, n, and h (the h value to use for the `der` function), and returns a function representing the n^{th} derivative of f.

3. **SICP Exercise 1.44** - The idea of smoothing a function is an important concept in signal processing. If f is a function and *dx* is some small number, then the smoothed version of f is the function whose value at a point x is the average of $f(x - dx)$, $f(x)$, and $f(x + dx)$.

   (a) Write a SCHEME procedure, named `smooth`, that takes as input a procedure that computes f and returns a procedure that computes the smoothed f.

   (b) It is sometimes valuable to repeatedly smooth a function (that is, smooth the smoothed function, and so on) to obtain the *n-fold smoothed* function. Write a SCHEME procedure, named `n-fold-smooth`, to generate the n-fold smoothed function of any given function using `smooth` and `repeated` from SICP Exercise 1.43 and this week's lab assignment.

   Remark 1. *In mathematics and computer science,* ***currying*** *is the technique of translating the evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument. You may find it helpful to define a curried function to use the same dx for each iteration of* `smooth`.

   ```
   (define (smoother dx) (lambda (f) (smooth f dx)))
   ```

4. **Ackermann Function**
   In lecture, we discussed "tail recursion" and how functions that are tail recursive can be converted in a straightforward way to an iterative solution using a looping structure. Functions of this type are referred to as *primitive recursive*. Unfortunately, not all recursive functions can be expressed using tail recursion. These are functions that simply must be expressed recursively. One such function is the famous Ackermann Function, named for Wilhelm Ackermann whose doctoral advisor was David Hilbert (a famous mathematician). Ackermann's function, with some subsequent refinement is generally expressed today as:

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

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
```

5. Let f and g be two functions taking numbers to numbers. We define $\mathbf{m}_{fg}$ to be the function so that

$$\mathbf{m}_{fg}(x) = \text{the larger of } f(x) \text{ and } g(x).$$

For example, if $f(x) = x$ and $g(x) = -x$ then $\mathbf{m}_{fg}$ is the absolute value function. Define a function `max-fg` which takes a *pair* of functions, `f` and g, as an argument and returns the function $\mathbf{m}_{fg}$ as its value. (So the return value is a *function*: the function which, at every point x, returns the larger value of $f(x)$ and $g(x)$.)

6. Romberg's method is a procedure for numerical integration. That is, a method for estimating the numerical value for the definite integral of a function like the Riemann Sum (rectangle method) in the lecture slides. Figure 1 shows an example of the Trapezoid Rule.

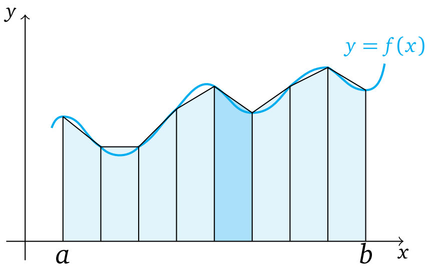


Figure 1: Example of numerical integration using the Trapezoid Rule.

Romberg's method is defined as follows:

$$R(n,m) = \begin{cases} h_1(f(a)+f(b)) & \text{if } n=0 \text{ and } m=0, \\ \frac{1}{2}R(n-1,0)+h_n\sum_{k=1}^{2^{n-1}} f(a+(2k-1)h_n) & \text{if } n \neq 0 \text{ and } m=0, \\ R(n,m-1)+\frac{1}{4^m-1}(R(n,m-1)-R(n-1,m-1)) & \text{otherwise.} \end{cases}$$

where $h_n = \frac{1}{2^n}(b-a)$.
Define a SCHEME function, named `(romberg f a b n m)`, which computes the definite integral using Romberg's method. In particular, `f` is the function we wish to integrate, `a` and `b` are the starting point and endpoint of the interval over which we wish to integrate (as shown in Figure 1), `n` indicates the number of points to use when computing the estimate, and `m` is indicates the extrapolation (see below).
The zeroth extrapolation, $R(n,0)$, is equivalent to the trapezoidal rule with $2n+1$ points and the first extrapolation, $R(n,1)$, is equivalent to Simpson's rule (see SICP Excercise 1.29) with $2n+1$ points. The second extrapolation, $R(n,2)$, is equivalent to Boole's rule with $2n+1$ points. This hints at a way to work incrementally on this problem.

Remark 2. *Notice, to compute the zeroth extrapolation, only the first two cases are needed in the definition for the function $R(n,m)$ defined above. It might be a good idea to code and test these two cases first, testing your solution with the zeroth extrapolation only. Then add the third case when you are reasonably sure the first two cases are correct.*

Remark 3. *You may find the generalized sum function from lab useful.*

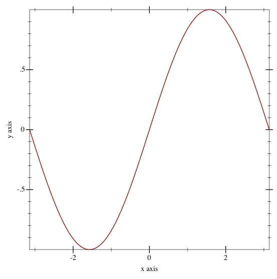
Try This at Home!

Once you have completed the deriviative (`der`) and n^{th} derivative (`der-n`) functions for the problem set, you can see these functions and your smoothing function in action. In a separate Racket file, you can use the Racket language to plot the sin function. Change the language to "Determine language from source" and place the following at the top of your new file:

```
#lang racket
(require plot)
```

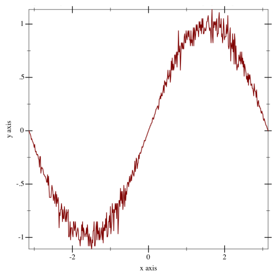
Now, you can plot the sin function:

```
(plot (function sin (- pi) pi))
```



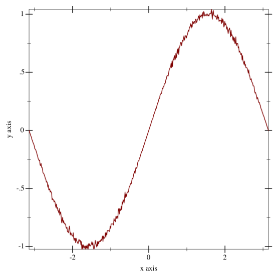
Next, copy over your derivative function solutions from the problem set to plot the n^{th} derivative of the sin function.

```
(define fourth-deriv-sin ((nth-deriv 4 0.00025) sin))
(plot (function fourth-deriv-sin (- pi) pi))
```



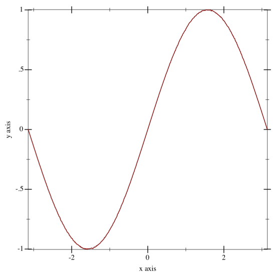
You should see something like this. Perhaps not what we expected. You can use the `smooth` function to reduce the noise, but it may not be enough.

```
(define (smoother dx) (lambda (f) (smooth f dx)))
(define smoothed ((smoother 0.00025) fourth-deriv-sin))
(plot (function smoothed (- pi) pi))
```



We can use the `n-fold-smooth` function to smooth out the fourth derivative of the sin function even further.

```
(plot (function (n-fold-smooth fourth-deriv-sin 0.00025 3) (- pi) pi))
```



Submit only your first Racket file (using the R5RS language) to Mimir.