MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.037—Structure and Interpretation of Computer Programs
IAP 2019

Project 1

Release date: 10 January, 2019
Due date: 15 January, 2019 at 1900h

In this project you will write more sophisticated procedures, including higher-order procedures and list manipulation. This project is divided into **two parts, which you should submit together**.

Part 1: Numerical integration reloaded

Problem 1: Integrating any function

In Project 0, you wrote procedures to integrate the function from Ben Bitdiddle's dream. Since we might want to approximate the area under *other* curves, let's **generalize that code**. We will pass the function to integrate as a parameter. You may implement `integral` using either a recursive or iterative algorithm.

```
(define (integral func num-steps x1 x2)
    'your-code-here)
```

Problem 2: Area of a unit circle

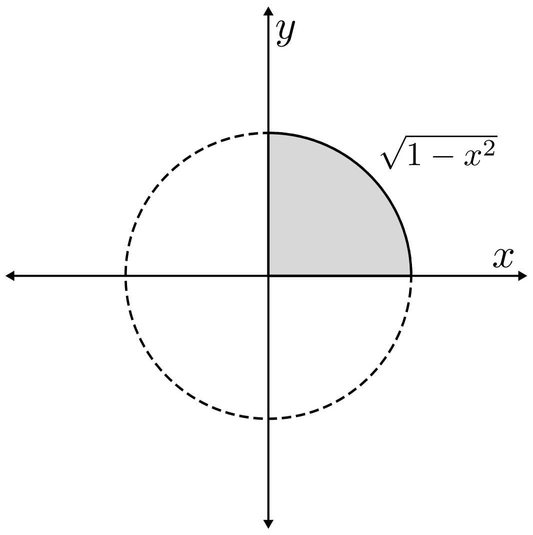
Integration is good for more than just dream interpretation. Recall that a circle of radius 1 is the set of points (x, y) such that

$$x^2 + y^2 = 1.$$

With a little algebra, we get

$$y = \pm\sqrt{1 - x^2}.$$

The area of our circle is four times the area of the upper-right quarter-circle:



Therefore,

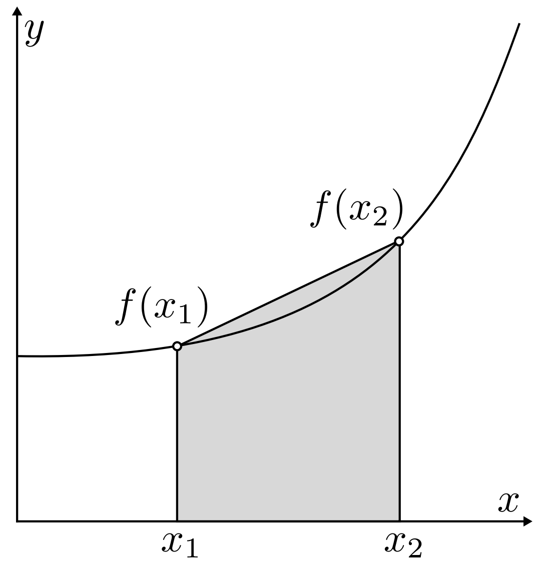
$$\pi = 4\int_0^1 \sqrt{1-x^2}\ \mathrm{d}x.$$

Write a procedure to approximate π using `integral`. Try to do it without any additional `define`s.

```
(define (approx-pi num-steps)
    'your-code-here)
```

Problem 3: Integrating with pieces of any shape

Alyssa P. Hacker points out to Ben that a rectangle is a rather poor approximation of a curve. What's better than a rectangle with one corner on a curve? A trapezoid with *two* corners on the curve!



Recall that `integral` sums a bunch of areas obtained from rectangles, like `bitfunc-rect` did for Ben's original function. We could change `integral` to use trapezoids – but instead, let's make this another parameter. **Write procedures `rectangle` and `trapezoid` to compute the area of the respective shapes. Then write `integral-with`, which is like `integral`, but takes one of these procedures as its first argument, and uses it for each segment of the integral.**

```
(define (rectangle func x1 x2)
    'your-code-here)

(define (trapezoid func x1 x2)
    'your-code-here)

(define (integral-with piece func num-steps x1 x2)
    'your-code-here)
```

The idea is that `(integral-with rectangle f n x1 x2)` should give exactly the same result as `(integral f n x1 x2)`, while `(integral-with trapezoid f n x1 x2)` might be more accurate (for the same `num-steps`).

Problem 4: Better approximation of π

Rewrite your procedure from Problem 2 to use trapezoidal integration.

```
(define (better-pi num-steps)
    'your-code-here)
```

Optional problems

There are many more sophisticated ways to do numerical integration. You could replace the trapezoids with quadratic functions; this is known as Simpson's rule. Or you could discard the idea of evenly-spaced intervals, and instead subdivide intervals whose estimated error is too high. If you like, implement one of these ideas, or another fancy integration method. Tell us what you did, and how much it improves accuracy.

Racket includes a sophisticated graphing system, which you can read about at `http://docs.racket-lang.org/plot`. Use this to illustrate some properties of your numeric integrators. For example, you could plot a function along with rectangular and trapezoidal area approximations, shading each one differently.

Part 2: Symbolic differentiation

Alyssa P. Hacker isn't too impressed with Ben's numerical integrals. "What good is an *approximate* answer? I'll get an exact answer, by manipulating *algebraic expressions containing variables.*" Alyssa does some reading and discovers that symbolic integration is very hard, so she decides to tackle symbolic differentiation.

Alyssa needs a way to represent algebraic expressions as Scheme data. She decides to represent numerical constants as Scheme numbers, and variables as Scheme symbols.

She starts with a procedure to find the derivative of a constant with respect to ("`wrt`") some variable:

```
(define (deriv-constant wrt constant)
    0)
```

And she tests it:

```
> (deriv-constant 'x 3)
0
```

"That was easy."

Problem 5: Derivative of a variable

Write a procedure to find the derivative of a variable with respect to some variable.

```
(define (deriv-variable wrt var)
    'your-code-here)
```

Test cases:

```
(deriv-variable 'x 'x)  ; -> 1
(deriv-variable 'x 'y)  ; -> 0
```

Problem 6: Calling the right function

So far, an expression could be either a constant or a variable. **Write a procedure which finds the derivative in either case**, by calling `deriv-constant` or `deriv-variable` as appropriate.

```
(define (derivative wrt expr)
    (cond
        ; your code here
        (else (error "Don't know how to differentiate" expr))))
```

Test cases:

```
(derivative 'x  3)  ; -> 0
(derivative 'x 'x)  ; -> 1
```

Problem 7: Derivative of a sum

Alyssa decides to represent the sum of two expressions as a 3-element list: the symbol `+` followed by the two expressions.

Recall that the derivative of a sum is just the sum of the derivatives:

$$\frac{d}{dx}(A+B) = \frac{d}{dx}A + \frac{d}{dx}B$$

And **write a procedure which finds the derivative of such an expression.**

```
(define (deriv-sum wrt expr)
    'your-code-here)
```

Then **modify your code for Problem 6 to call `deriv-sum` when appropriate**. You can add it to your solution for Problem 6, as long as you make a note here. Test case:

```
(derivative 'x '(+ x 2))  ; -> (+ 1 0)
```

Problem 8: Derivative of a product

Alyssa represents products similarly, using `*` instead of `+`. Recall the product rule for derivatives:

$$\frac{d}{dx}(A \cdot B) = A \cdot \frac{d}{dx}B + \frac{d}{dx}A \cdot B$$

And then **write a procedure which finds the derivative of a product, and extend `derivative` as before.**

```
(define (deriv-product wrt expr)
    'your-code-here)
```

Test case:

```
(derivative 'x '(* x 3))  ; -> (+ (* x 0) (* 1 3))
```

Returning any algebraically equivalent expression is also fine.

Problem 9: Additional testing

Write some additional test cases to test all the functionality of `derivative`.

Optional problems

- Improve your code to handle sums and products of more than two expressions.
- Add support for more kinds of algebraic expressions.