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

Project 0

Release date: 8 January, 2019
Due date: 10 January, 2019 at 1900h

Due to the fast-paced nature of this class, we want you to get some practice right away with writing and testing Scheme procedures. These exercises might not take a lot of time, but we want to make sure you are prepared for Project 1, which builds on these ideas.

If you have problems completing everything, please submit whatever you manage to finish, along with a note at the top mentioning what you completed, what doesn't work, and where you think the problem lies. Don't hesitate to ask for help by sending the instructors mail at `6.001-zombies@mit.edu`. We also encourage you to work with others on projects as long as you acknowledge it. If you cooperated with other people, please indicate your consultants' names and how they collaborated in your submission.

Read the entire project description below before you start working. Every phrase in **boldface** describes something that you should include in your final submission — e.g., procedures to write, test cases to run, or questions to answer.

You should create a file with your solutions, which you will email to us. For each problem below, **include your code**, and demonstrate its functionality against a set of test cases. You should always **create and include your own meaningful test cases** (even when we provide some) to ensure that your code works not only on typical inputs, but also on more difficult cases. Get in the habit of writing and running these test cases after *every* procedure you write — no matter how trivial the procedure may seem to you.

You should also include **comments explaining what's going on** in your code. A good comment does not simply restate in English what the code already says. Rather, it explains *why* that code exists, in terms of the problem being solved. You should also use comments to **identify which problem** you are solving, by number. In Scheme, a comment starts with a semicolon and continues to the end of the line.

This project includes some provided code in the file `simple-integration.scm`, which can be obtained from the *projects* link on the course web page.

Preparing and submitting your solutions

If you are using the DrRacket program, you can write your procedures in the *definitions* window. When you click the *Run* button, these definitions become available in the *interactions* window. When you type Scheme code into this window and press Enter, DrRacket will print the result. You can use this to evaluate your procedures on some test cases.

One easy way to show us your test cases is to copy your interactions into the definitions window as a comment. To do this, select your interactions by pressing the left mouse button with the cursor positioned at one end, then drag the mouse to the other end and release the button. This will highlight the text, which you can then copy with Ctrl-C. If you then click the mouse on the *definitions* window, you can insert the copied text into that window by placing the cursor and typing Ctrl-V. Now highlight the text you just inserted, click on the *Racket* menu, and click on *Comment*

out with semicolons. This converts the text into a comment, so that the Scheme interpreter will ignore it when you press *Run*. Be sure not to comment out the procedures themselves!

When you are finally ready to submit your code, you can save the definitions as a **text mode file** and **email it to the course staff** at `6.037-psets@mit.edu`.

Problem 1: Bitdiddle's Function

One day, Ben Bitdiddle had a dream about the function

$$f(x) \ = \ x^4 - 5 * x^2 + 4.$$

Implement Bitdiddle's Function as a Scheme procedure.

```
(define (bitfunc x)
    'your-code-here)
```

Hint: the Scheme syntax for x^y is `(expt x y)`.

Problem 2: A rectangle under Bitdiddle's Function

Ben's dream left him with the ineffable desire to compute the area under various parts of this curve. One crude approximation is to find the area of a single rectangle which extends from the x-axis to the curve. Ben decides (somewhat arbitrarily) that the height of the rectangle will be determined by the value of his function at the *left* x-coordinate.

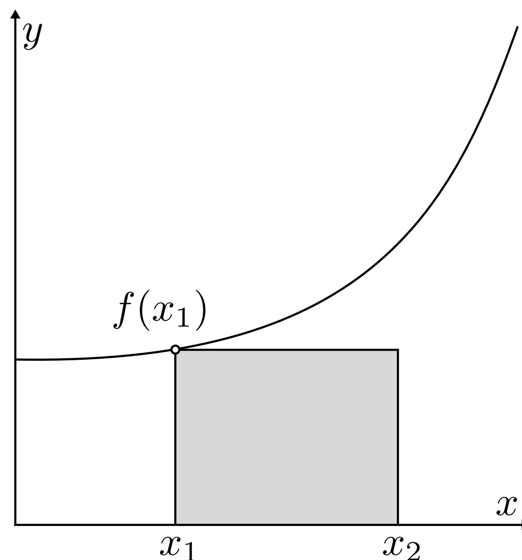

Figure 1: Approximating the area between `x1` and `x2` using one rectangle

Write a Scheme procedure to find the area of such a rectangle, given two x-coordinates. You may assume `x1` $\leq$ `x2`.

```
(define (bitfunc-rect x1 x2)
    'your-code-here)
```

Problem 3: Integrating Bitdiddle's Function

One rectangle gives a poor approximation, but the sum over many small rectangles should be better. **Write a Scheme procedure to find the total area of a number of small rectangles**, with each rectangle of equal width, and the appropriate height for the function at that point.

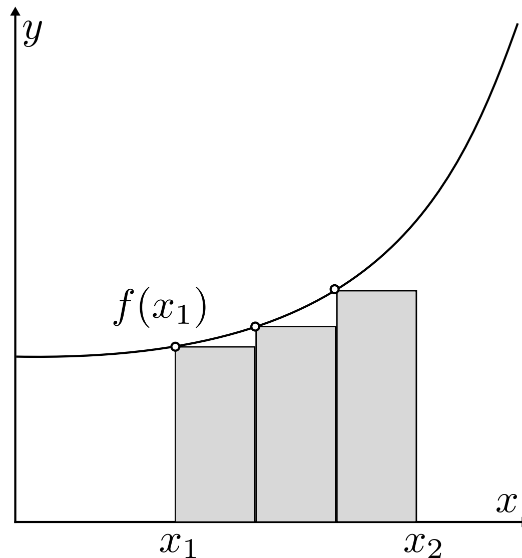

Figure 2: Slightly better approximation using three rectangles

Do this **once using a recursive algorithm** and **again using an iterative algorithm**. See the lecture 1 slides for a description of what this means.

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

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

Problem 4: Comparing the two integrators

Write a procedure that returns the absolute value of the difference between the results of the two integrator procedures, for a given `num-steps`, `x1`, and `x2`. This should always be very close to zero. Due to the inexact nature of floating-point arithmetic, it might not be exactly zero.

Hint: We didn't tell you the name of the Scheme procedure for absolute value, but you can find it with a quick web search.

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