**CS1101: Lab 4 – Using Lists to Create Analog to Digital Converters**

(Acknowledgement: thank you to Allison Smyth, who developed these exercises as part of an IQP)

Most microcontrollers have analog to digital converters, which are used to convert everyday signals that appear in nature (analog) to signals that can be used by a microprocessor (digital). Most analog to digital converters provide several bit resolutions from which one can be chosen for a conversion. The bit-resolution used in a conversion is an important factor that decides the accuracy of the conversion. In this lab, we will be creating analog to digital converters using structures and lists in Racket.

**Lab Motivation and Goals**

* Write templates over lists
* Program with lists
* Work with lists embedded in structs

**Exercises**

An electrical engineer has decided that he would like to add digital systems to his original toolbox of analog devices. Before he is able to do this, he needs to build analog to digital converters (ADCs) so that his old electrical devices will function with his new ones. An analog to digital converter consists of a processor name (string) as well as a list-of-number. This list represents the bit-resolutions that the ADC can use for a conversion. The data definition for an analog to digital converter is given below. Copy this into your Lab 4 file and also create a data-definition for list-of-number before proceeding with the lab.

 ;; an ADC is a struct

 ;; (make-ADC string list-of-number)

 (define-struct ADC (processor bits))

1. Write the template for processing ADCs (you will need two templates, one for ADCs and one for a list-of-number).
2. Create an example of an ADC with bit resolutions of 4, 8, 10, and 12.
3. Write a function **best-processor** that consumes two ADCs and produces a string representing the name of the processor whose ADC has the largest number of bit resolutions. (Hint: You should create a helper function to count the number of items in the list.)
4. Write a function **average** that consumes an ADC and produces a number representing the average bit-resolution of the ADC. (Hint: Always try to reuse previous functions when possible. Be careful not to waste time writing the same code twice.)
5. Write a function **improve-ADC** that consumes an ADC and produces an ADC with the same processor; however, every bit-resolution of the original ADC is incremented by 2.
6. Write a function **add-res** that consumes an ADC and a number representing a bit-resolution. It should produce an ADC with the new bit-resolution added to the current list of bit-resolutions unless the inputted ADC already has this bit-resolution or better (a bit-resolution higher than the input).

**Everybody should be able to finish up to this point.**

1. Write a function **bit-levels** that consumes an ADC and produces a list-of-number representing resolution levels the ADC is capable of producing. A resolution level is equal to 2 to the power of a bit-resolution. (Note: Racket has a function called *expt* that consumes two numbers and produces a number. The function raises the first number to the power specified by the second number.)
2. Write a function **remove-odds** that consumes an ADC and produces an ADC with all odd bit resolutions removed. (Note: We are doing this because it is very rare for a processor to use an odd-bit resolution for a conversion). (Hint: Racket has a function called *modulo* that consumes two numbers and produces the remainder of the division of the first number by the second.)

**Wow you’re fast! These are a little more challenging, but are good extra practice.**

1. This problem will be completed in two-steps. We wish to create a function that will produce an ADC that is good for performing conversions with a set of accelerometers. All accelerometers will produce a .3 volt change on their output line when the sensor detects a change of 1g. Therefore, at 0g it outputs 0V, 1g outputs .3V, and so on. Some accelerometers have different output voltage ranges than others. This affects the conversion value received from that accelerometer.
2. Write a function **ADC-Value** that consumes a list-of-number representing bit-resolutions of an ADC and an accelerometer max output voltage (number) and produces a list-of-number representing the ADC conversion result for a 1g change on the accelerometer. This equation may help: $ADC Value= \frac{.3\*2^{n}}{Input Voltage Range}$ (where n = bit-resolution value).
3. Write a function **remove-bad-res** that consumes an ADC and an accelerometer max output voltage (number) and produces an ADC with all bit-resolutions that cannot be used in conversions with this accelerometer removed. (Hint: A bit-resolution is not appropriate when it produces a result less than 1 since this means the ADC is not accurate enough to display the necessary information.)
4. This problem will also be completed in two-steps. We want to determine if all bit-resolutions of an ADC are capable of performing the conversion of a certain input signal with error less than a specified amount.
5. Write a function **get-error** that will consume a list-of-number representing the bit-resolutions of an ADC and a number representing the maximum value of an input signal to the ADC. This function should produce a list-of-number that represents the maximum error for each bit-resolution. This equation may help:$ Max Error= \frac{Input Voltage Range}{2^{n+1}}$.
6. Write a function **good-ADC?** that consumes an ADC, a number representing the maximum value of the input signal, and a number representing the maximum error wanted. It produces true if all bit-resolutions produce a max error value less than the maximum error inputted to the system. It produces false otherwise.
7. Write a function **sort** that consumes an ADC and produces an ADC such that all bit-resolutions in the new ADC are sorted from smallest to largest. (This function requires a different approach than the others. We will be covering it later in class, but give it a try and see what you come up with).