CSC 3205: Structure and Interpretation of Computer Programs

Take Home Assignment

April 2022

Instructions

1. To be handed in via MUELE on the link provided.
2. Deadline is Friday 22^{nd} April 2022, 5:00pm.
3. This is an individual assignment.

Part 1: Scheme Vectors

In this assignment, we look at vectors. Scheme vectors are indexed from 0 to n-1, where n is the number of elements. Unlike lists, elements in vectors can be directly accessed. Like we did for lists and pairs, we write procedures that operate on vectors.

1. Implement a function **sum-vector** that takes two vectors of numbers as arguments and returns a vector with the sum of the corresponding elements of the input vectors e.g.,

   ```
   (sum-vector (vector 4 6 8 3) (vector 5 6 7)) => #(9 11 15 3)
   ```

   Note that the two input vectors are not required to be of the same length.

2. Consider the following vector **colors**, which contains the names (strings) of colors.

   ```
   (define colors
      (vector "red" "orange" "yellow" "green" "blue" "indigo" "violet"))
   ```

   Write a function **length-of-vector-elements** that takes such a vector as an argument and returns a vector of lengths of the elements e.g.,

   ```
   (length-of-vector-elements colors) => #(3 6 6 5 4 6 6)
   ```

   Hint: Scheme provides **string-length** function that returns a length of a string.

3. a. Recall the **map** function in Scheme that takes a function and a list of elements and returns a list of results of applying the function on each element. Implement a function **vector-map** that does the same for vectors e.g,

   ```
   (vector-map (lambda (x) (* x x)) (vector 1 2 3 4)) => #(1 4 9 16)
   ```

   b. Similarly, define a function **vector-for-each** that works like **for-each** for lists e.g.,

   ```
   (vector-for-each display (vector "red" "orange")) => "red""orange"
   ```

Part 2: Variadic Functions

Up to now we have been writing functions that take a fixed number of arguments. But of course as you know there are cases where the number of arguments are not known before hand. Take an example, the + native Scheme function, can be invoked with a variable number of arguments e.g.,

```
(+ 10) => 10
(+ 10 20) => 30
(+ 10 20 40) => 70
```

In this part of the exercise, we explore how we can define such functions.

1. Define a function **display-all**, which is a variation of the native function **display**. **display-all** takes one or more arguments and displays them.

   ```
   (display-all "foo") => "foo"
   (display-all "foo" "bar") => "foo" "bar"
   ```

2. Define a function **sum-them-all** that takes one or more arguments and returns their sum e.g.,

   ```
   (sum-them-all 10 20 30 40) => 100
   (sum-them-all 12 8 2) => 22
   ```

3. Using the the function **sum-them-all** defined in (2), define a function **average** that takes one or more arguments and returns their average e.g.,

   ```
   (average 10 20 30 40) => 25
   (average 10 20 30) => 20
   (average 10) => 10
   ```

4. Define a function **maximum-of-many** that takes as argument two or more numbers and returns the maximum one. e.g.,

   ```
   (maximum-of-many 10 20 30 40) => 40
   (maximum-of-many 12 8) => 12
   ```

Good Luck!