UNIVERSITY OF ALBERTA

Augustana Computing Science 370

Assignment #4

Goals:

- To program successfully in the Scheme programming language.
- To grasp concepts of functional programming, and know why it is important in other paradigms.
- To write small, powerful functions, and to see the value of this technique in programming.
- To be able to use recursion well.

References:

Scheme resources from the course outline.

Instructions:

Write a set of Scheme functions that operate on vectors. A vector will be in index-value format that is a list of elements and each element contains an index (0-based) and the corresponding value. All values are non-zero, except in the case of the last index, which may have a value of 0, but indicates the maximum index in the vector. The indices must be in strictly ascending order. The empty list is the null vector, a vector with length 0.

For example:

```
((0 3) (4 13) (5 7) (8 0))
```

is the vector

```
(3, 0, 0, 0, 13, 7, 0, 0, 0)
```

```
((3 15))
```

is the vector

```
(0, 0, 0, 15)
```

And

```
((5, 0))
```

is the vector

```
(0, 0, 0, 0, 0, 0)
```

[General Computing Science information: This representation for vectors is particularly useful when many of the entries are expected to be 0. It is called sparse-representation.]

The set of functions that you write must include the following:

1) **printVector** — Takes a vector and prints it out in standard form, with all zeros filled in and commas separating elements. This function does not print any spaces or newlines, either before or after the vector. This function also does not create a new list – it only prints the vector. For example:

```
> (display "hi") (printVector '((0 1) (1 4) (3 -2) (10 65) (11 0))) (display "hi")
hi(1, 4, 0, -2, 0, 0, 0, 0, 0, 0, 65, 0)hi
```

2) **iPrintVector** — Takes a vector and prints it in indexed form, with only non-zero and final entries being printed. This function ends the vector with a newline. For example:

```
> (iPrintVector '((0 1) (1 4) (3 -2) (10 65) (11 0)))
x0 = 1
x1 = 4
x3 = -2
x10 = 65
x11 = 0
```

3) **addVectors** — Takes two vectors and returns a vector which is their sum. The addition is done on a positional basis, and if one vector is longer that becomes the length of the sum (under the assumption that the shorter vector is filled with zeros in the extra places). For example:

```
> (addVectors '((0 1) (1 4) (3 -2) (10 65) (11 0)) '((1 -4) (3 -1) (10 0)))
((0 1) (3 -3) (10 65) (11 0)
```

Note that values which become zero must be eliminated from the sum, unless it is the last index.

4) **subVectors** — Takes two vectors and returns a vector which is their difference, i.e. the first vector minus the second. The subtraction is done on a positional basis, and if one vector is longer that becomes the length of the sum (under the assumption that the shorter vector is filled with zeros in the extra places). For example:


```
> (subVectors '((0 1) (1 4) (3 -2) (10 65) (11 0)) '((1 -4) (3 -1) (10 0)))
((0 1) (1 8) (3 -1) (10 65) (11 0))
```


Note that values which become zero must be eliminated from the sum, unless it is the last index.

5) **scaleVector** — Takes a scalar and a vector, and returns the vector with all values multiplied by the scalar. In the case of the scalar zero, only the last index will remain. For example,


```
> (scaleVector -1 '((4 5) (8 7) (9 -2)))
((4 -5) (8 -7) (9 2))
```

6) **dotProduct** — Takes two vectors and returns the dot product of the two vectors. If the two vectors are different sizes, returns false (`#f`). For example,

```
> (dotProduct '((1 3) (2 4) (5 2)) '((0 6) (1 2) (3 7) (5 8)))
22
```

7) **avValue** — Takes a vector and returns the average of the values in the vector, taking into account the full length of the vector. The result is in decimal form.

```
> (avValue '((2 2) (4 18)))
4.0
```

8) **maxValue** — Takes a vector and returns the maximum value in the vector. For example,

```
> (maxValue '((0 1) (1 4) (3 -2) (10 65) (11 0)))
65
```

9) **minValue** — Takes a vector and returns the minimum value in the vector. For example,

```
> (minValue '((0 1) (1 4) (3 -2) (10 65) (11 0)))
-2
```

10) **maxIndex** — Takes a vector and returns the maximum index in the vector. For example,

```
> (maxIndex '((2 2) (4 18) (15 0)))
15
> (maxIndex '())
#f
```

11) **size** — Takes a vector and returns the size of the vector, which is one more than the maximum index or 0 in case of the empty vector. For example,

```
> (size '((2 2) (4 18) (15 0)))
16
> (size '())
0
```


Some items to consider in your implementation:

1) You must use a functional programming style. In particular, the only use of sequencing is for the purpose of output (there is no input in this program) and the only use of assignment is with `define` for a function name.

2) See eClass for a list of functions that are allowed in your program. You must limit yourself to this list, together with any other functions that you write. This list is the same as for the previous assignment.

3) The only exception to rules 1) & 2) is for the purpose of testing, where you may use define and set! to your heart's content.
4) You may define any other helper functions, as needed.
5) Your function names must match the requirements exactly, otherwise they will not work with my testing driver. I cannot be changing any names, and your testing results will then fail.
6) No terms with zero values should be left in any vectors, except where that is the maximum index.

Marking:

Due to the large number of students in our Computing Science program, it is no longer possible for the instructor to run each program individually. This means that your code (+ testing) must speak for itself, and convince the instructor that your program works. You must hand in testing of your program. The instructor will run some programs each time. These will be chosen at random or if it seems the code does not match the testing submitted. If the code does not match the testing that is submitted, then a mark of 0 (zero) will be awarded for the assignment.

Considerations when marking:

1) Correctness (must be in Scheme, working, no plagiarism, no contorted or unnecessary code, testing submitted)
2) Documentation (Easy to read and follow, indentation, file header [name, date, summary], program subdivision, subprogram headers [summary, parameter, return], descriptive variable names including method names, no useless code, inline useful comments [no code translation],

main steps are clear [pop out], recursion levels reasonable, organization)

3) Style (includes design, efficiency, appropriate/idiomatic use of the Scheme programming language)

A program with syntax errors will attain a mark of 50% at best, depending on documentation, style, and how much of the program appears finished.

Hand In:

1) Your Scheme file (both as a code file .rkt and as a .pdf).
2) A .pdf of your testing, run with the testing file provided on eClass.

Credit:

This assignment has been adapted from a similar one by Anna Koop.