HW4: Scheme #1

Due Midnight 3/1/17

All procedures should be defined in a single source code file (using the extension `.scm`). Procedure names and parameters must match up with the requirements from this document. Check the file on UTC Learn that describes ways to work with Guile.

Your file must load into guile with no errors to be eligible for any points. I will be loading it like this:

```
bash> guile -l file.scm
```

Each procedure is worth 10 points.

Do not use any imperative functions.

1 `(yourname)`

This procedure should return your name. Everyone's procedure will do something slightly different!

For example:

```
scheme@(guile-user)> (yourname)
$60 = "Craig Tanis"
```

Please note, you will lose points if your procedure returns `"Craig Tanis"`.

2 `(ax+b a x b)`

Return the mathematical value of $ax + b$.

For example:

```
scheme@(guile-user)> (ax+b 10 20 30)
$54 = 230
scheme@(guile-user)> (ax+b 1 1 1)
$55 = 2
```

3 (distance p1 p2)

Assuming `p1` and `p2` correspond to points in the x-y plane, return the Euclidean distance between the two points. You know Pythagoream, right?

For example:

```
scheme@(guile-user)> (distance '(0 0) '(1 1))
$58 = 1.4142135623730951
scheme@(guile-user)> (distance '(10 0) '(10 100))
$59 = 100
```

4 (purge match lst)

Return a list containing the items in `lst` that are not equal (using the `eq?` procedure) to `match`.

For example:

```
scheme@(guile-user)> (purge 'bat '(Na na na na na na na na bat man))
$56 = (Na na na na na na na na man)
scheme@(guile-user)> (purge 'na '(Na na na na na na na na bat man))
$57 = (Na bat man)
```

5 (count-trues lst)

This procedure should return the count of the number of items in `lst` that are considered true. Use the built-in `filter` procedure, and the built-in `length` procedure.

For example:

```
scheme@(guile-user)> (count-trues '(#f #f #f #f '() ok))
$51 = 2
scheme@(guile-user)> (count-trues '(1 2 3 socks #f))
$52 = 4
scheme@(guile-user)> (count-trues '())
$53 = 0
```

6 (build-list n)

This procedure should use recursion to return a new list of integers from `n` to 0 (exclusive).

For example:

```
scheme@(guile-user)> (build-list -5)
$19 = (-5 -4 -3 -2 -1)
scheme@(guile-user)> (build-list 3)
$20 = (3 2 1)
scheme@(guile-user)> (build-list 0)
$21 = ()
```

Note: this has nothing to do with the Scheme built-in function `make-list`.

7 (dotproduct v1 v2)

Given two lists, `v1` and `v2`, make sure that they both have the same length (see the built-in procedure `length`). If there is a size mismatch, return `#f`. Otherwise, pretend the lists are vectors and return their dot product. Suggestion: Use `map` to add up the elements pairwise, and `apply` the `+` procedure to sum them.

For example:

```
scheme@(guile-user)> (dotproduct '(1 2 3) '(100 10 1))
$24 = 123
scheme@(guile-user)> (dotproduct '(1 2 3) '(100))
$25 = #f
```

8 (multiples base n)

This procedure should return all multiples of `base` between `(* n base)` and 0 (exclusive). This should work regardless of whether or not the arguments are positive or negative. Use your `build-list` procedure!

For example:

```
scheme@(guile-user)> (multiples 1 10)
$34 = (10 9 8 7 6 5 4 3 2 1)
scheme@(guile-user)> (multiples -5 12)
$35 = (-60 -55 -50 -45 -40 -35 -30 -25 -20 -15 -10 -5)
scheme@(guile-user)> (multiples 12 -5)
$36 = (-60 -48 -36 -24 -12)
```

Tips:

- Don't worry about what happens if `base` is 0.
- This is an easy application of `map`.

9 (run-cmd opname lst)

This function applies a function to `lst` that is determined by looking up `opname`.

Consider this application of the `cond` form that uses a symbol to determine a message:

```
(define (lookup-msg tag)
  (cond ((eq? tag 'hello) "Hello world")
        ((eq? tag 'goodbye) "Goodnight, Moon")
        ((eq? tag 'scuzz) "You are a scuzz bucket")
        (else (string-append "Unknown command: "
                                  (symbol->string tag)))))
```

In this particular case, `opname` should correspond to the following symbols:

- `'plus` – applies the `+` procedure
- `'times` – applies the `*` procedure
- `'append` – applies the `string-append` procedure
- `'cdr` – applies the `cdr` procedure
- If none are matched, simply return `lst`.

For example:

```
scheme@(guile-user)> (run-cmd 'plus '(1 2 3))
$41 = 6
scheme@(guile-user)> (run-cmd 'times (build-list 4))
$42 = 24
scheme@(guile-user)> (run-cmd 'append '("foo" "bar" " " "jones"))
$43 = "foobar jones"
scheme@(guile-user)> (run-cmd 'not-a-thing '(x y z))
$44 = (x y z)
scheme@(guile-user)> (run-cmd 'cdr '(1 2 buckle my shoe))
$45 = (2 buckle my shoe)
```

10 (charflip str)

Assumes a string argument, and returns a string where uppercase characters are tranformed to lowercase, and vice versa.

You should refer to (and use) the following built-in procedures:

- `char-upper-case?`
- `char-downcase`
- `char-upcase`
- `list->string`

- `string->list`

Transform the string to a list, `map` some function over the list, and then pack the list back into a string.

For example:

```
scheme@(guile-user)> (charflip "The Fat Boys are back")
$46 = "tHE fAT bOYS ARE BACK"
scheme@(guile-user)> (charflip "ZzZzZz")
$47 = "zZzZzZ"
scheme@(guile-user)> (charflip "1 2 3 Hack ALERT!")
$48 = "1 2 3 hACK alert!"
```