COSC 455
Programming Languages: Design and Implementation
Fall 2017

Project #2 – Functional Programming

Directions: This project will culminate in you completing a few small programs using functional Scala and Scheme. Please read through the entire project description before starting it.

Goals: The intention of this project is to provide continued experience in functional programming and compare the implementation Scala to that a pure functional language (Scheme) and that of imperative and logic programming languages. In doing so, you should develop a better understanding of functional programming and the concepts behind this class of programming languages.

This project will give you development experience in the following areas of programming languages that were covered in class:

- Functional programming
- Scala
- Scheme
- List data structures
- Anonymous and higher-order functions

Further, the implementation of these concepts will give you further experience in essential programming concepts including:

- Problem solving
- Recursion
- Program design

Deadline: Tuesday, November 28, 2017, 11:59pm to Blackboard

Language Requirements: GitHub, the IntelliJ IDEA Community Edition IDE and using Scala and SBT must be used in this Project along with Slack (#project2 channel) to communicate and submit portions of the lab. No other environment, programming language or submission will be allowed. *Only functional aspects of Scala may be used in any solution*.

The DrRacket environment (http://racket-lang.org/) and compiler should be used for Scheme. The DrRacket/Scheme environment is freely available to download and install on various platforms at http://download.racket-lang.org/. To configure the DrRacket environment to use the correct version of Scheme, make sure "Determine Language from Source" is noted in the bottom left corner of the environment window and include `#lang racket` as the first line in all your Scheme programs.

Grading: There is a maximum total of 43 points and will be scored out of a possible of 38 points. The grading for this project is as follows:

- The Scala Warmup Programs are a series of 4 problems to solve each with explicitly marked points for a correct solution for a total of 5 points.
- The Binary Addition/Subtraction Program will be worth 8 points as follows: 5 points allocated for code development, 3 points for correct solution.
- The Scala Translation Program will be worth 10 points as follows: 6 points allocated for code development; 2 points for correct solution; and 2 points for quality of your code (i.e., structure, comments, etc.).
- The Scheme Warmup Programs are a series of problems to solve each with explicitly marked points for a correct solution for a total of 6 points.
- The Scheme Translation Program will be worth 10 points as follows: 6 points allocated for code development; 2 points for correct solution; and 2 points for quality of your code (i.e., structure, comments, etc.).
- The Drunk Homer Program has a possible 5 points available for extra credit towards your project grade.

Submission: All projects must submit a single zip file to Blackboard containing the IntelliJ Scala Worksheet files and Scheme source code for each program. Please label your source files so that they can be graded easily. *Failure to submit the project follow these guidelines may result in your project not being graded.*

I. Scala Warmup (5 points)

Please go through the following exercises to prepare for the functional Scala implementations of two, somewhat larger Scala programs. Each of these exercises will be beneficial for binary addition and translation program and your understanding of these exercises will make the translation program easier and quicker to implement.

1. *Prime Numbers* (1 points). Create a function, named *prime*, that takes an integer and returns a Boolean indicating whether the integer parameter is a prime number. A prime number is an integer greater than 1 that has no positive divisors other than 1 and itself.

2. *Twin Primes* (1 points). Create a function, named *twinprimes*, that takes 2 integer parameters and returns a Boolean indicating whether the parameters are twin primes. A twin prime is a prime number that differs from another prime number by two, for example the twin prime pair (41, 43). For example, *twinprimes (41, 43)* should return *true* and *twinprimes (43, 47)* would return *false*.

3. *Twin Primes List* (1 points). Create a function, named *twinprimeslist*, that takes an integer, *n*, parameter and returns an integer list of all the twin primes starting up to *n*. For example, *twinprimeslist (50)* should return *[3, 5, 7, 11, 13, 17, 19, 29, 31, 41, 43]* (no duplicates).

4. *Goldbach's Conjecture* (2 points). Create a function, named *goldbach*, that takes an integer and prints the solution satisfying the Goldbach Conjecture. The Goldbach Conjecture states that every positive even number greater than 2 is the sum of two prim numbers. For example, 28 = 5 + 23. Your function is to find the two prime numbers that sum up to a given even integer and print the composition. For example *goldbach(28)* would print *5 + 23 = 28*. You should provide error checking to make sure the integer parameter is even and greater than 2.

Some helpful hints:

- You may need to write helper functions in some solutions that are not explicitly named here.
- You should use recursive solutions, no looping/iterating allowed!
- Recall the way that pattern matching in a list works in Scala – this should be very helpful in a good solution. For full credit, you should use pattern matching over if-else statements where possible.
- As in any language, adopt a "write once, use often" mentality. If there is something you need to do several times, write a function for it and use it appropriately.
- Test each function you write incrementally within the Scala worksheet to make sure it works correctly.

II. Binary Addition/Subtraction (8 points)
Functional programming's use of lists to simulate/model digital circuits is common. For this phase of the project, we will develop the functions to perform binary addition given two lists of binary integers (i.e., 1s and 0s). Consider the basic binary addition rules:

0 + 0 = 0
1 + 0 = 1
0 + 1 = 1
1 + 1 = 10
1 + 1 + 1 = 11

Consider the following example:

```
  11110
+  1011
----------
 101001
```

Binary addition works from right to left. Two bits plus the carry bit (if necessary from the right) give a sum bit for this position and a carry to the left. Just as with normal decimal addition, when the sum in one column is a two-bit (two-digit) number, the least significant figure is written as part of the total sum and the most significant figure is "carried" to the next left column. Consider the following examples:

```
                        11  1   <--- Carry bits ----->     11
  1001101             1001001                       1000111
+ 0010010           + 0011001                     +   10110
 -- ---------        ------------                  ------------
  1011111              1100010                      1011101
```

You are to develop the functions in Scala to implement binary addition utilizing pattern matching as much as possible and *without using any built-in arithmetic functions* from the skeleton code I have provided. From the provided Scala Worksheet, the binaryAddition function should take two integer lists of binary integers (i.e., 0s and 1s) and perform the binary addition with the doBinaryAddition function. I have defined 4 test cases (from the above examples) that you can use to test your final code.

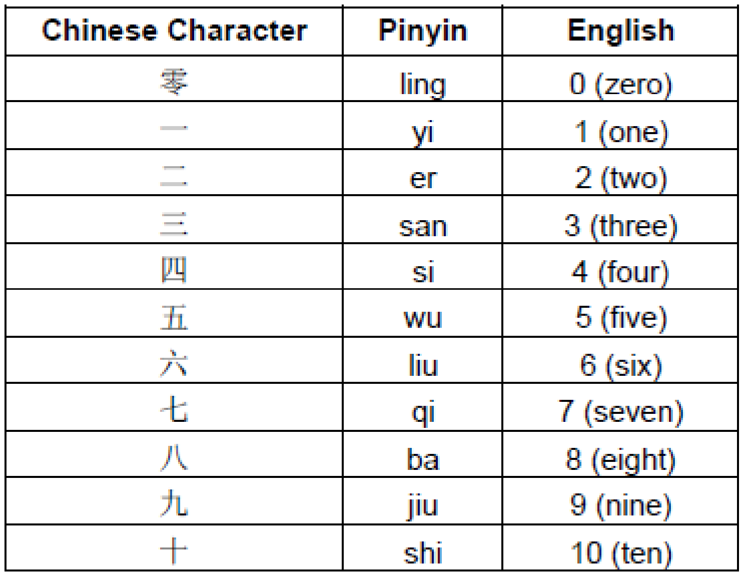
Since list processing functions work best from left to right (i.e., head to tail) and binary addition is done from right to left (i.e., tail to head), you should first reverse the list (you may use Scala's built in reverse function. To make the computation easier, you should also convert the lists from binary integers to Boolean values (you may assume that all input values are valid).

Binary addition can be performed on lists of unequal length. If one of the bit lists terminates before the addition is complete (as determined by the doBinaryAddition function), then the addition and carry bit should be propagated with the other bit list in the finishBinaryAdd function.

Additionally, develop a binary subtraction function, named *binarySubtraction*, that takes in two lists and performs the binary subtraction. Hint: recall how binary subtraction similar to binary addition after taking the two’s compliment.

III. Language Math Translation - Scala (10 points)

Hanyu pinyin is the most commonly used Romanization system for the Standard Mandarin Chinese language. *Hanyu* means “Chinese language” and *pinyin* means “spell sound” or “phonetic”. Consider the following table:



Chinese Character	Pinyin	English
零	ling	0 (zero)
一	yi	1 (one)
二	er	2 (two)
三	san	3 (three)
四	si	4 (four)
五	wu	5 (five)
六	liu	6 (six)
七	qi	7 (seven)
八	ba	8 (eight)
九	jiu	9 (nine)
十	shi	10 (ten)

Use this table to construct lists of Chinese and English words. For example, a list written in Scheme as:

```
val chinese: List[String] = List("ling", "yi", "er", "san", "si", "wu", "liu", "qi", "ba", "jiu", "shi")
val english: List[String] = List("zero", "one", "two", "three", "four", "five", "six", "seven", "eight", "nine", "ten")
```

As one who has studied Mandarin Chinese, I would like a Scala program that would work with an input list of numbers written in English and/or Chinese pinyin, adds and multiplies the list. For example, I should be able to query within your Scala Worksheet with a function `go` as follows:

```
go(List(‘yi’,‘nine’,‘six’,‘ba’))
```

which should produce the output (in the correct order!):

```
Translation: 1 9 6 8
Addition: 1 + 9 + 6 + 8 = 24
Multiplication: 1 * 9 * 6 * 8 = 432
```

Your program, however, should be able to filter out numbers it doesn’t understand. For example, if I entered:

```
go(List(‘yi’,‘josh’,‘three’,‘si’))
```

your program should discard `josh` and output:

```
Translation: 1 3 4
Addition: 1 + 3 + 4 = 8
Multiplication: 1 * 3 * 4 = 12
```

Some helpful hints:

- You may only use `head`, `tail` and `::`. You may **not** use the built-in `take` function or any other Scala-defined functions.
- Recall that in class we previously developed a `member` function that takes an element and checks whether it is a member of a list.
- Recall the way that pattern matching in a list works in Scala – this should be very helpful in a good solution.
- As in any language, adopt a "write once, use often" mentality. If there is something you need to do several times, write a function for it and use it appropriately.
- Test each function you write incrementally within the Scala worksheet to make sure it works correctly.

IV. Scheme Preparation (10 points)

Please go through the following exercises to prepare for the Scheme implementation of the Language Translation program. Each of these exercises will be beneficial for translation program and your understanding of these exercises will make the translation program easier and quicker to implement.

Experiment with ' (quote) in the Scheme interpreter. Try, for example 'e, '+, '(+ 3 4), 'cosc455, and unquoted versions of expressions. When you understand what ‘ (quote) does, type in the following code to Scheme:

```
(define alist '())
(define anotherlist '(a b c))
(cons (car anotherlist) alist)
(cons (car (cdr anotherlist)) alist)
(display alist)
```

After understanding what cons does, does this code do what you expected? Why or why not? After answering the above question, try the following code:

```
(define alist '())
(define anotherlist '(a b c))
(set! alist (cons (car anotherlist) alist))
(set! alist (cons (car (cdr anotherlist)) alist))
(display alist)
```

Make sure you understand what set! does. Notice the order in which Scheme appends to a list. This may not always be what we’d like. Modify the last line to be (display (reverse alist)) and execute it.

Apply Scheme's car and cdr functions to a list that you define. Be sure to show before and after views of lists that you apply car and cdr to. (Hint: you will need to use define.) Make sure to understand how car and cdr work – we’ve seen the equivalent functions in Scala already when working with lists!

Now try the following in Scheme:

```
(define capitals '((maryland (annapolis)) (pennsylvania
(harrisburg))(delaware (dover)) (virginia (richmond))))
(car (cdr capitals))
(cadr capitals)
(cadar capitals)
(caadar capitals)
```

Do you see what Scheme is doing here with cadr, cdar, caadar, etc.?

1. (1 point) Some Scheme developers prefer to have more descriptive functions for car and cdr. Given this, consider the following functions that use anonymous functions (called Lambda functions in Scheme):

```
(define first (lambda (x) (car x)))
```

```
(define second (lambda (x) (cadr x)))
(define rest (lambda (x) (cdr x)))
```

and then the following interactions:

```
(define family '(josh sara erin sandy jon))
(first family)
(second family)
```

Once you understand what this code does, develop functions for third, fourth and fifth and include them in your submission.

2. (1 point) Develop a function, named *truecount*, that takes a list of booleans (i.e., `#t` or `#f`) and returns the number of trues (i.e., `#t`) in the list and include it in your submission.

3. (1 point) As in Scala, Scheme supports anonymous functions, called Lambda functions in Scheme as well as the map function. For example, the following code in Scheme, entered in the interactions window, that uses both map and a Lambda function to add 1 to each item in the list:

```
(define intlist (sqrlst alist))
(map (lambda (x) (+ x 1)) intlist))
```

   With this, develop a function, named squarelist, that takes a list and squares each item of the list and include it in your submission.

4. (1 point) Scheme additionally provides a higher-order function, named filter, that applies a function to a list to decide if it should keep the item or not. Try the following code in the interactions window:

```
(filter positive? '(1 4 -1 6 4 -7))
```

   With this, develop a function, named hundreds?, that takes a list and returns any integer greater than 100. To do so, use Scheme's filter function and an anonymous function in your hundreds? function. Include the function in your submission.

5. (2 point) Develop a function, named *collatz*, that takes an integer, *n*, and prints out the hailstone sequence. The hailstone sequence is determined as follows: if *n* is even, the function divides it by 2 to get *n* / 2, if *n* is odd multiply it by 3 and add 1 to obtain 3*n* + 1. Repeat the process until you reach 1. Hints: 1. In Scheme, you can print out an integer, n, followed by a newline as follows (display n) (newline); and, 2. Scheme has handy built-in functions odd?, even? and eq?. Include the function here along with a few screenshots testing the function.

V. Math Translation - Scheme (10 points)

Hanyu pinyin is the most commonly used Romanization system for the Standard Mandarin Chinese language. *Hanyu* means “Chinese language” and *pinyin* means “spell sound” or “phonetic”. Consider the following table:

Chinese Character	Pinyin	English
零	ling	0 (zero)
一	yi	1 (one)
二	er	2 (two)
三	san	3 (three)
四	si	4 (four)
五	wu	5 (five)
六	liu	6 (six)
七	qi	7 (seven)
八	ba	8 (eight)
九	jiu	9 (nine)
十	shi	10 (ten)

Use this table to construct lists of Chinese and English words. For example, a list written in Scheme as:

```
(define chinese ‘(ling yi er san si wu liu qi ba jiu shi))
(define english ‘(zero one two three four five six seven eight nine ten))
```

As one who has studied Mandarin Chinese, I would like a Scheme program that would work with and input list of numbers written in both English and Chinese translates, adds and multiplies the list. For example, I should be able to query within Dr Racket’s interaction panel with a function `go` as follows:

```
(go ‘(yi nine six ba))
```

which should produce the output (in the correct order!):

```
Translation: 1 9 6 8
Addition: 1 + 9 + 6 + 8 = 24
Multiplication: 1 * 9 * 6 * 8 = 432
```

Your program, however, should be able to filter out numbers it doesn’t understand. For example, if I entered:

```
(go ‘(yi josh three si))
```

your program should discard `josh` and output:

```
Translation: 1 3 4
```

Addition: 1 + 3 + 4 = 8
Multiplication: 1 * 3 * 4 = 12

Some helpful hints:

- Don't forget the `first, second`, etc. functions from the preparation phase, the `member` function from the slides and how `filter` works – this may be helpful!
- Recall the `cond` function acts like a switch statement. This may be helpful for looking up the translations.
- As in any language, adopt a "write once, use often" mentality. If there is something you need to do several times, write a function for it and use it appropriately.
- The debug facility in DrRacket is very helpful, use it!

Scheme Resources

There are a number of excellent resources on the Scheme language that you may wish you use, including:

- *The Scheme Programming Language* by R. Kent Dybvig http://www.scheme.com/tspl2d/
- *Teach Yourself Scheme* http://ds26gte.github.io/tyscheme/

VI. Extra Credit - Drunk Homer (5 points)

Homer Simpson, after a stop at Moe's, went to the Springfield Mall to buy Marge, Lisa, Bart and Maggie a gift in anticipation that they will be upset with him when he gets home. While at the mall, Homer buys 4 gifts: a green dress for Marge, a saxophone book for Lisa, a slingshot for Bart and a new pacifier for Maggie. He recalls buying the gifts at the following stores: The Leftorium, Sprawl-Mart, Try-N-Save, and King Toots.

Somewhere along the way, Homer lost his car keys and had to walk home carrying the gifts. Wanting to retrace his steps and find his lost car keys, the family asks Homer where he bought the gifts and the order in which he bought the gifts. Being partly inebriated however, Homer can't remember which stores he bought the gifts at and in which order he visited the stores. After some sobering up, Homer remembers the following:

- He bought the saxophone book at King Toots
- The store he visited immediately after buying the slingshot was not Sprawl-Mart
- The Leftorium was his second stop
- Two stops after leaving Try-N-Save, he bought the pacifier

Develop a Scala program that will help Homer solve the mystery of the stores where he bought each gift and the order in which he bought them. For this question only, you may use any features/built-in functions of Scala and are not restricted to only functional Scala. You may not, however, hardcode the solution into your program.