

Project #2 – Functional Programming

Directions: This project will culminate in you completing a small program in another functional programming language, Scheme, and consists of two phases: the first phase will allow you to discover, learn and answer questions about Scheme's implementation; the second phase will utilize what you've learned to design and implement a small program in 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 to 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
- Scheme, SML
- 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: Saturday, November 28, 2015, 11:59pm to Blackboard

Language Requirements: The Standard ML of New Jersey (SML/NJ) compiler for the Standard ML '97 (<http://www.smlnj.org/sml97.html>) programming language should be used for SML. The SML/NJ compiler is open source and freely available to download and install on various platforms at <http://www.smlnj.org/dist/working/110.72/index.html>.

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: The grading for this project is as follows:

- The SML Binary Addition Program will be worth 10 points as follows: 6 points allocated for code development, 4 points for correct solution and 2 points possible of extra credit for completing the binary subtraction function.
- The Scheme Preparation phase is worth 10 points total.

- 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.).
- There are 5 points available for extra credit towards your project grade.

Submission: All projects must submit: 1. a text file (e.g., .docx, .odt) with your answers, screenshots (when required) and source code (when required) from the Scheme preparation phase; 2. source code for the programs; and, 3. screenshots of example output. Submit all files as a zip file to Blackboard. *Failure to submit the project follow these guidelines may result in your project not being graded.*

SML Binary Addition (10 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:

$$\begin{array}{r} 11110 \\ + 1011 \\ \hline 101001 \end{array}$$

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:

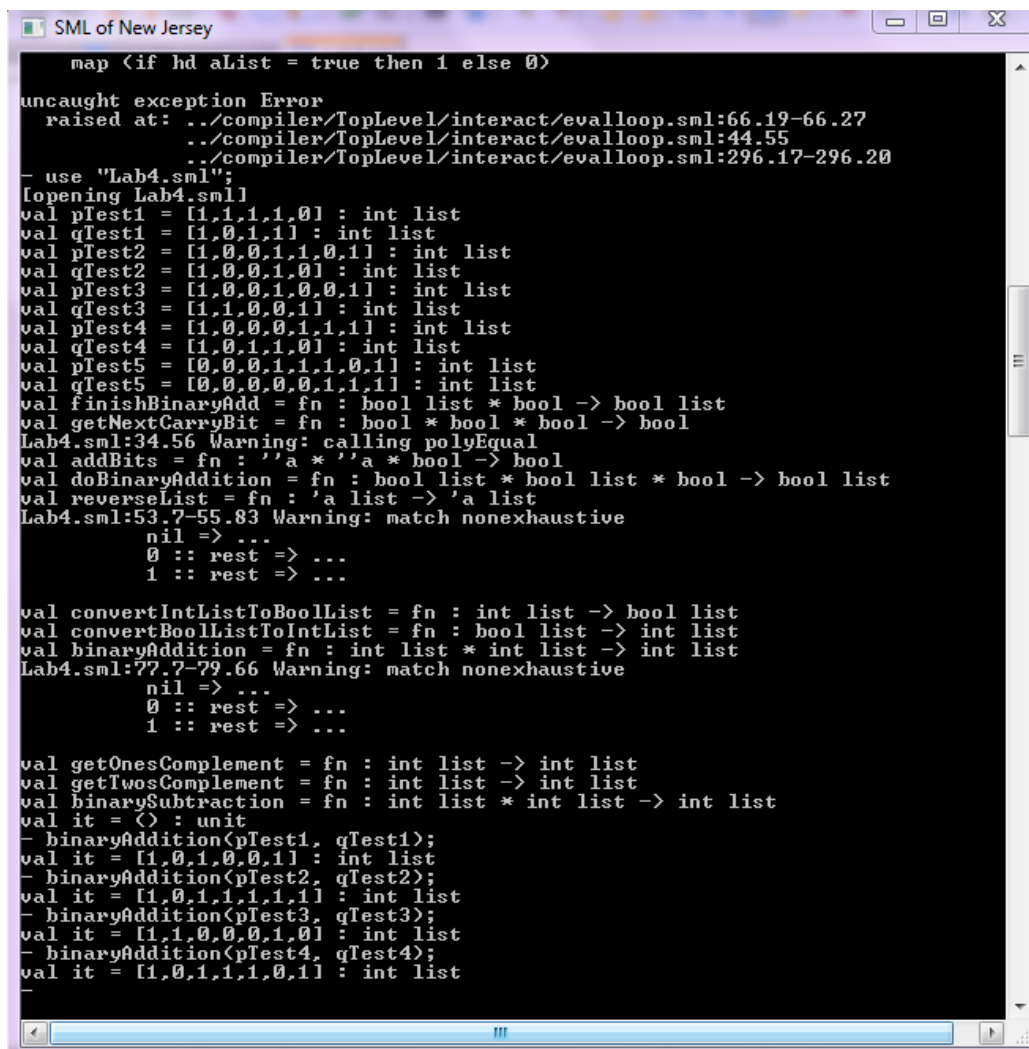
	11 1 <--- Carry bits -----> 11	
1001101	1001001	1000111
+ 0010010	+ 0011001	+ 10110
-- -----	-----	-----
1011111	1100010	1011101

You are to develop the functions in SML 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 SML command line, 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 SML's 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. I have provided a `reverseList` function to do this. 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.

To test your fully developed function, you should use the test cases I have defined for you. A set of correct answer is provided in the screen shot of my solution below.



```

map <if hd aList = true then 1 else 0>

uncaught exception Error
  raised at: ../compiler/TopLevel/interact/evalloop.sml:66.19-66.27
             ../compiler/TopLevel/interact/evalloop.sml:44.55
             ../compiler/TopLevel/interact/evalloop.sml:296.17-296.20
- use "Lab4.sml";
[opening Lab4.sml]
val pTest1 = [1,1,1,1,0] : int list
val qTest1 = [1,0,1,1] : int list
val pTest2 = [1,0,0,1,1,0,1] : int list
val qTest2 = [1,0,0,1,0] : int list
val pTest3 = [1,0,0,1,0,0,1] : int list
val qTest3 = [1,1,0,0,1] : int list
val pTest4 = [1,0,0,0,1,1,1] : int list
val qTest4 = [1,0,1,1,0] : int list
val pTest5 = [0,0,0,1,1,1,0,1] : int list
val qTest5 = [0,0,0,0,0,1,1,1] : int list
val finishBinaryAdd = fn : bool list * bool -> bool list
val getNextCarryBit = fn : bool * bool * bool -> bool
Lab4.sml:34.56 Warning: calling polyEqual
val addBits = fn : 'a * 'a * bool -> bool
val doBinaryAddition = fn : bool list * bool list * bool -> bool list
val reverseList = fn : 'a list -> 'a list
Lab4.sml:53.7-55.83 Warning: match nonexhaustive
      nil => ...
      0 :: rest => ...
      1 :: rest => ...

val convertIntListToBoolList = fn : int list -> bool list
val convertBoolListToIntList = fn : bool list -> int list
val binaryAddition = fn : int list * int list -> int list
Lab4.sml:77.7-79.66 Warning: match nonexhaustive
      nil => ...
      0 :: rest => ...
      1 :: rest => ...

val getOnesComplement = fn : int list -> int list
val getTwosComplement = fn : int list -> int list
val binarySubtraction = fn : int list * int list -> int list
val it = () : unit
- binaryAddition(pTest1, qTest1);
val it = [1,0,1,0,0,1] : int list
- binaryAddition(pTest2, qTest2);
val it = [1,0,1,1,1,1,1] : int list
- binaryAddition(pTest3, qTest3);
val it = [1,1,0,0,0,1,0] : int list
- binaryAddition(pTest4, qTest4);
val it = [1,0,1,1,1,0,1] : int list
-

```

Extra Credit (2 points)

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.

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.

1. (0.5 point) 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, provide a short description on its function.
2. (0.5 point) 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)
```

Briefly describe 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)).

3. (0.5 point) Provide a transcript (code and listing from the interactions panel) showing that Scheme's car and cdr procedures do not change their arguments. Be sure to show before and after views of lists that you apply car and cdr to. (Hint: you will need to use define.) Provide a short description on how car and cdr work.
4. (0.5 point) What do each of the following return in Scheme?
 - (define capitals '((maryland (annapolis)) (pennsylvania (harrisburg))(delaware (dover)) (virginia (richmond))))
 - (car (cdr capitals))
 - (cadr capitals)
 - (cdar capitals)
 - (cadar capitals)
 - (caadar capitals)
 - (cdadar capitals)
 - (car (cdadar capitals))

Briefly describe the functionality of cadr, cdar, caadar, etc.

5. (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 here along with some screenshots testing the functions.

6. (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.
7. (1 point) As in SML, 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 (sqrlist 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 the function here along with a few screenshots testing the function.

8. (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 here along with a few screenshots testing the function.

9. (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 $3n + 1$. 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.

10. (2 points) Develop a function, named *fizzbuzz*, that takes a single integer parameter and solves the fizzbuzz problem from 1 to n. That is, develop a function that prints the numbers from 1 to n and for multiples of 3 print “Fizz” instead of the number and for the multiples of 5 print “Buzz”. For numbers which are multiples of both three and five print “FizzBuzz”. Hints: 1. The modulus function in Scheme is `modulo`; and, 2. There is a built in function, `zero?`, that takes in a single integer and returns `#t`/`#f` depending on if it is zero or not. Include the function here along with a few screenshots testing the function.

Scheme Program – Language Math Translation (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 an 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/>

Extra Credit SML Program – Language Translation (5 points)

Develop the Language Translation program, described above, in SML. To gain full credit, you should utilize the following in SML:

- Pattern matching
- Recursion
- User-defined data types
- Exception handling