CptS355 - Assignment 3 (Scheme)
Spring 2017

Scheme Functions

Assigned: Sunday February 19, 2017

Due: Wednesday March 8, 2017

Weight: Assignment 3 will count for 6% of your course grade.

Your solutions to the assignment problems are to be your own work. Refer to the course academic integrity statement in the syllabus.

This assignment provides some practice in Scheme programming. You will write Scheme functions (and test functions) for the given problems. You will use the Racket environment, which can be downloaded from http://racket-lang.org. After installing Dr. Racket, be sure to select R5RS as the language on the language menu. You can edit your code in the upper panel of the Racket IDE and save it from there. Or edit in any text editor and load it into Racket. (Make sure to save your file with .scm extension.)

Turning in your assignment

Turn in a plain text file named HW3.scm containing legal scheme code -- you should be able to take the contents of your file and run it through racket without errors. Create your own test cases for each function and provide test functions comparing the output of your testcases to the correct output.

To submit your assignment, turn in your file by uploading on the Assignment3 (Scheme) DROPBOX on Blackboard (under AssignmentSubmisions menu). Be sure to include your name as a comment at the top of the file. You may turn in your assignment up to 5 times. Only the last one submitted will be graded. Please let the instructor know if you need to resubmit it a 6th time.

The work you turn in is to be your own personal work. You may not copy another student's code or work together on writing code. You may not copy code from the web, or anything else that lets you avoid solving the problems for yourself.

Grading

The assignment will be marked for correctness and effective programming. You will lose points : (1) if your functions give errors for certain test cases, (2) if they don't produce the correct output, (3) if you don't provide test functions, and (4) if you don't explain your code with appropriate comments.

General rules

Unless directed otherwise, you should implement your functions using recursive definitions that you build up from the basic built-in functions such as `cons, car, cdr, list`, etc. Don't use set! and don't define anything except functions.

Problems

1. (deepSum L) – 15%

The function `deepSum` takes a single argument, `L`, a list, and returns the sum of elements (atoms) contained in `L` and the sublists in `L`. You can assume that list L contains either integers or sublists containing integers.

Example:

```
(deepSum '(1 (2 3 4) (5) 6 7 (8 9 10) 11))  returns 66.
(deepSum '( () ))  returns 0.
```

(Note: `deepSum` should check whether each element in L is a list or not. Remember that the predicate function `(pair? L)` returns true if L is a pair value (such as a list). `(pair? '()) returns #f` (i.e., empty list is not a pair value.))

2. (numbersToSum sum L) - 15%

The function `numbersToSum` that takes an `int` (called `sum`) (which you can assume is positive), and an `int list L` (which you can assume contains positive numbers) and returns a `list`. The returned list should include the first `n` elements from the input list such that the first `n` elements of the list add to less than `sum`, but the first (`n + 1`) elements of the list add to `sum` or more.

Assume the entire list sums to more than the passed in `sum` value.

Examples:

```
(numbersToSum 100  '(10 20 30 40))  returns (10 20 30)
(numbersToSum 30  '(5 4 6 10 4 2 1 5))  returns (5 4 6 10 4)
```

3. **(isSorted L) - 15%**

The function `isSorted` takes a list (which contains positive numbers) and returns true if the elements in the list are in ascending order. `isSorted` will return false otherwise. (Hint: use `length` function to check the length of a list)

Examples:

```
(isSorted '(1 4 5 6 10))  returns #t
(isSorted '(1 3 6 5 10))  returns #f
(isSorted '(1))  returns #t
```

4. (mergeUnique2 L1 L2) - 10%

The function `mergeUnique2` takes two lists of integers, `L1` and `L2`, each already in ascending order, and returns a merged list that is also in ascending order and that doesn't include any duplicates. The resulting list should be the union of the elements of the two lists. Duplicates should be eliminated during the merge. You may assume that input lists, `L1` and `L2`, don't contain any duplicates.

Examples:

```
(mergeUnique2 '(4 6 7)  '(3 5 7)) returns (3 4 5 6 7 )
(mergeUnique2 '(1 5 7) '(2 5 7)) returns (1 2 5 7)
(mergeUnique2 '() '(3 5 7)) returns (3 5 7)
```

(fold L)

Just use the definition given in class. You'll need it for the function below.

5. (mergeUniqueN Ln) – 10%

a) Using `mergeUnique2` function defined above and the fold function, define `mergeUniqueN` which takes a list of lists, each already in ascending order, and returns a new list containing all of the elements in ascending order. (You may assume that the sublists in `Ln` don't contain any duplicate values.)

(Provide an answer using fold; without using recursion).

Examples:

```
(mergeUniqueN '()) returns ()
(mergeUniqueN '((2 4 6) (1 4 5 6))) returns (1 2 4 5 6)
(mergeUniqueN '((2 4 6 10) (1 3 6) (8 9))) returns (1 2 3 4 6 8 9 10)
```

Note that here `fold` returns a list, not just a simple value; but the solution is, in fact, a very simple use of `fold` once you choose the right value for the base case corresponding to the empty list as input.

b) In a comment, discuss the question of how many `cons` operations your function uses to produce its result, in terms of the sizes of the input lists. Explain your answer. For this problem, I suggest looking first at how many `cons` operations are used by `mergeUnique2` for lists of length `len1` and `len2`. If `mergeUniqueN` is used for a list with a single sublisy, what is the answer? For 2 sublists? For n sublists?

6. (matrixMap f M) – 10%

A matrix M can be represented in Scheme as a list of lists, for example `M='((1 2) (3 4))`
Without using recursion, write a function `matrixMap`, which takes a function `f` and a matrix `M` as arguments and returns a matrix consisting of `f` applied to the elements of `M`.
Examples:

```
(matrixMap (lambda (x) (* x x)) '((1 2) (3 4)) ) returns ((1 4) (9 16))
(matrixMap (lambda (x) (+ 1 x)) '((0 1 2) (3 4 5)) ) returns
((1 2 3) (4 5 6))
```

7. (avgOdd L) – 10%

Without using recursion, write a function `avgOdd` which takes a list `L` and returns the average of the odd values in `L`.
Examples:

```
(avgOdd '(1 2 3 4 5)) returns 3
(avgOdd '(1 3 5)) returns 3
(avgOdd '(1 2 4 6)) returns 1
```

Use (odd? n) and (even? n) predicate functions to check if a given number n is odd or even.

8. (unzip L) – 15%

In class we defined a function named `zip` that takes two lists as its arguments and produces a list of pairs as its output. For this problem, define the inverse of `zip` without using recursion, namely `unzip`, that takes a list of two-element lists as input and produces a list of two lists as output. (Hint: Call the `map` function twice to get the first and second values from pairs)
What should `(unzip '())` be?
Examples:

```
(unzip '((1 2) (3 4) (5 6))) returns ((1 3 5) (2 4 6))
(unzip '((1 "a") (5 "b") (8 "c"))) returns ((1 5 8) ("a" "b" "c"))
```

Scheme resources:

- R5RS (the standard)
- How to Design Programs (with links to the DrScheme web page)
- Structure and Interpretation of Computer Programs
- Programming Languages: Application and Interpretation