CS59-F16 Homework 2

Terms and Conditions. This homework is open-book, open-shell, open-Internet (in your submission, make a note of the resources you used).

Honor Principle: You may discuss the homework with other students at a high level (tools and techniques) on the class list, or on Slack at `https://cs59-fall16.slack.com/`, but should never look at or copy another student's written work. **You should not turn in the homework that is a duplication or partial duplication of another student's solution. You must also credit other students you have talked with, or any other sources.**

Language: The intended language for these exercises is LISP. Use Emacs LISP, GCL, or SBCL; ask if you want to use another dialect/implementation of LISP. For all problems LISP-y recursive style is expected throughout, and LISP is the most natural way of working with input s-expressions.

What to submit: Submit your work as a **syntactically valid** LISP file named `hw2-DDDD.l` where `DDDD` are the last four symbols of your ID. Submit the file by emailing it as an attachment to me. Use LISP comments `;` to supply explanations on your solutions; see `https://www.gnu.org/software/emacs/manual/html_node/elisp/Comment-Tips.html` for preferred comment style.

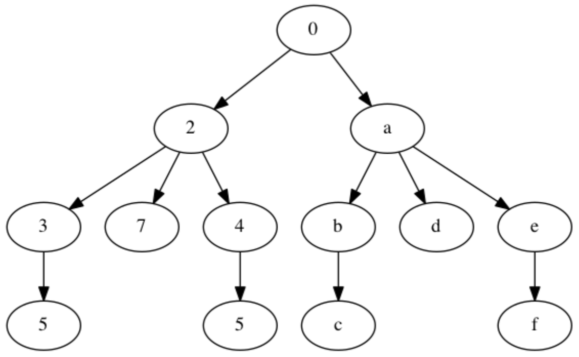
Make sure to name your functions as specified in the problems.

When to submit: by noon of Saturday, October 15th.

Introduction.

In these exercises, you are given s-expressions: LISP lists that can contain both atoms and other lists. You can think of an s-expression (also called a *sexp*) as a tree made up of cons cells, with atoms as leaves. List nesting can be arbitrarily deep (i.e., don't assume any fixed maximum depth of nesting).

There are several ways to represent a graphical tree structure with numbers, symbols, or strings at the nodes as a list. In Problem 3, we will be using the following way: at each level, the atom value inside the tree node will be the first element of the list, and any branches will be its siblings, in order from left to right. The example below demonstrates this encoding:



will be represented as

```
(0 (2 (3 (5)) (7) (4 (5))) (a (b (c)) (d) (e (f))))
```

The input to Problem 3 in this homework is assumed to be given in the above format only.[1]

[1]Please note that there are other ways of representing graphical trees as sexps in LISP; e.g., the above tree could be written as `(0 (2 (3 5) 7 (4 5)) (a (b c) d (e f)))`, with leaves written as atom siblings of the parent, saving some cons cells. But for Problem 3 we are only using this particular representation.

Problem 1. *"Specialis Revelio"*

(a) Write a function `(revelio test list)` that takes a sexp `list` and a predicate `test` applied to the atom value at the node, and returns the **first** item in **depth-first traversal** order that satisfies the test and nil otherwise.

Our test cases will make use of multiple predicates.[2] Note that the item can reside deep in the tree; think of the nodes as Gringotts vaults, and of the sexp as the map of the vaults. You need to come up with a stored item that matches the test, at any fork first exploring the path that leads deeper down.

For example:

```
(revelio 'evenp nil)
=> nil

(revelio 'evenp '(1 3 (5 (7)) 9))
=> nil

(revelio 'evenp '(1 3 (5 (8)) 9 10))
=> 8

(revelio 'zerop '(1 3 (5 (8)) 0 10))
=> 0
```

(b) Write the function `(revelio-viam test list)` that takes the same arguments, and returns a LISP program (as a sexp), that would produce that item when evaluated on the input list, or nil if no item satisfying the test exists in `list`.

For example:

```
(revelio-viam 'evenp nil)
=> nil

(revelio-viam 'evenp '(1 3 (5 (7)) 9))
=> nil

(revelio-viam 'evenp '(1 3 (5 (8)) 9 10))
=> (lambda (l) (car (car (cdr (car (cdr (cdr l)))))))
; (lambda (l) (caadr (caddr l))) is also correct.

(revelio-viam 'zerop '(1 3 (5 (8)) 0 10))
=> (lambda (l) (car (cdr (cdr (cdr l)))))
; (lambda (l) (cadddr l)) is also correct.
```

Problem 2. *Horcruxes begone!*

A horcrux is... Never mind what it is, you will have a predicate `horcruxp` that tells you if its argument is one. You want to keep any horcruxes from appearing in your return values.

Write functions to perform the following:

- A function `(flatten-depth-first horcruxp list)` that takes a sexp `list`, and returns a list of the sexp's atoms that are not horcruxes, in depth-first traversal order. For example,

[2] `http://www.tutorialspoint.com/lisp/lisp_predicates.htm`

```
(flatten-depth-first 'zerop nil)
=> nil

(flatten-depth-first 'evenp '(10 3 (5 (7)) 9))
=> (3 5 7 9)

(flatten-depth-first 'oddp '(10 3 (5 (7)) 9))
=> (10)

(flatten-depth-first 'evenp '(10 2 (4 (9)) 9))
=> (9 9)
```

- A function `(flatten-breadth-first horcruxp list)` that takes a sexp `list` and returns a list of its atoms that are not horcruxes, in breadth-first traversal order. You will need to use a queue; extra credit if your code uses no explicit loops. For example,

```
(flatten-breadth-first 'zerop nil)
=> nil

(flatten-breadth-first 'evenp '(1 3 (5 (7)) 9))
=> (1 3 9 5 7)

(flatten-breadth-first 'evenp '(1 2 (5 (7)) 9))
=> (1 9 5 7)
```

Problem 3. *"Geminio" & "Mobiliarbus"*

- Write a function `(tree-of-sums list)` that takes a sexp `list` representing a tree as described in the introduction, as argument, and returns a new tree of exactly the same shape, with each node containing the sum of all numbers in the original subtree under it (including the root of the subtree). Treat other kinds of atoms as zeros. For example,

```
(tree-of-sums nil)
=> nil

(tree-of-sums '(1 (3 (5 (7))) (9)))
=> (25 (15 (12 (7))) (9)))

(tree-of-sums '(1 (3 (a (7))) (c)))
=> (11 (10 (7 (7))) (0)))
```

- Write a function `(sorted-tree-bfs list)` that takes a tree `list` with numbers at all nodes as an input and returns a tree of exactly the same shape, but with the same numbers sorted and appearing in its nodes in breadth-first order (i.e., BFS yields a sorted list). Use ascending order (root gets the smallest number). For example,

```
(sorted-tree-bfs nil)
=> nil

(sorted-tree-bfs '(9 (3 (5 (1))) (7)))
=> (1 (3 (7 (9))) (5))
```

Suggestion: start with small examples that you can easily draw and check. Write simple sanity-check functions to test your results.

- **Extra credit:** Write a function `(verify-tree list)` that takes a sexp `list` and returns true `t` if the sexp is a valid tree in our representation, and `nil` otherwise. If given an optional key argument `:test`, your function should additionally check that all leaves in the tree satisfy the test. An empty tree is always OK. For example,[3]

```
(verify-tree nil)
=> t

(verify-tree '(1 2 3))
=> nil

(verify-tree '(1 (3) (a)))
=> t

(verify-tree '(1 (3) (a)) :test #'numberp)
=> nil
```

Problem 4. *"Computatio!"*

Write a function `(eval-arithm list)` that takes a sexp `list` (or an atom) that represents a valid LISP arithmetic expression using operators `+`, `-`, `*`, and `/`, evaluates it, and returns the evaluated numeric value.

Your function should return `:error` if the sexp passed as argument is *not* a valid arithmetic expression or cannot be evaluated. Assume that arithmetic operations `+`, `*`, `/` are **only** binary, and that `-` can be both binary and unary, i.e., `(- 3)` and `(- 2 5)` are both valid (and both evaluate to -3). Note that LISP's own `eval` allows the likes of `(* 3)` and `(+ 2)`, but yours shouldn't. Single numbers are OK, and should evaluate to themselves.

For example,

```
(eval-arithm '(+ (* 2 3) 7))
=> 13

(eval-arithm '(+ (* 2 3) (- 3 1)))
=> 8

(eval-arithm '(+ (* 2 3) (wingardium-leviosa 3 1)))
=> :error
```

[3]That spell would probably be called *"verifiarbus"*.

```
(eval-arithm '(+ (* 2 3)))
=> :error

(eval-arithm '(- "horcrux"))
=> :error

(eval-arithm 1001)
=> 1001
```

Extra credit: Write a function `(eval-arithm-with-vars list alist)` that takes an additional argument of the type called an *association list* (or *alist* in LISP jargon) that contains dotted pairs `(var . val)` that give values of one or more symbol variables. These symbol variables are allowed to occur in the arithmetic sexp, and should evaluate to their values given in the `alist`. An empty `alist` is permitted, but all variables occurring in the sexp `list` must also occur in the `alist`, otherwise you should return `:error`.

For example,

```
(eval-arithm-with-vars '(+ (* x 3) 7) '((x . 2)))
=> 13

(eval-arithm-with-vars '(+ (* 2 y) (- y zz)) '((zz . 1) (y . 3) (x . 1000)))
=> 8

(eval-arithm-with-vars '(+ (* 2 3) (wingardium-leviosa 3 1)))
=> :error

(eval-arithm-with-vars '(+ (* 2 3)) '((zz . 1001)))
=> :error

(eval-arithm-with-vars '(- "horcrux") '((zz . 1001)))
=> :error

(eval-arithm 1001 nil)
=> 1001

(eval-arithm '(zz) '((zz . 1001)))
=> 1001

(eval-arithm '(+ 1 zz) '((zz . 1001)))
=> 1002

(eval-arithm '(+ 1 zz) nil)
=> :error
```