# CISC-681/CISC-481/CGSC-681

**Assignment 1: 100 points**

**Due Friday, Sept. 23, 2016**

**CISC-481: 1,2,3,4,5: 13 pts. each; 7: 17 pts., 9: 18 pts.**

**CISC-681 and CGSC-681: 1,2,3,4,5: 6 pts. each; 6,7,8,9,10: 14 pts. each**

Design and test the following functions. You may not use any Lisp functions that essentially do the function for you, such as `sort`; also you must not use constructs such as `prog` and `loop` but should instead use recursion. If you have a question about whether you can use a particular Lisp function, please ask first to avoid problems. Near the due date, I will provide a set of test cases which you must execute and turn in, along with your code.

1. Design and test a Lisp function `most_once` that takes as argument two lists of integers and returns the number of elements of the first list that occur at most once in the second list.

```
(most_once '(6 4 3) '(7 6 2 6 4 4 5 3 2 3 4))      returns 0
(most_once '(8 3 2 3 5 7) '(7 3 4 5 8 3 3 5 4))      returns 3
```

2. Design and test a Lisp function `Cartesian_prod` that takes as argument two simple lists and returns their Cartesian product as a list of two-element lists. The resultant list should be ordered as shown in the examples below — i.e., the two-tuples constructed from the first number in the first argument list appear first in the result list, the two-tuples constructed from the second number in the first argument list appear next in the result list, and so forth.

```
(Cartesian_prod '(8 3) '(4 9)      returns      ((8 4)(8 9)(3 4)(3 9))
(Cartesian_prod '(2 5 3) '(8 4 2))      returns      ((2 8)(2 4)(2 2)(5 8)(5 4)(5 2)(3 8)(3 4)(3 2))
(Cartesian_prod '(34 55)(43 23 12)      returns      ((34 43)(34 23)(34 12)(55 43)(55 23)(55 12))
```

3. Design and test a function `duplicate_atoms` that takes as argument a list of integers and returns a list of sublists where the $i^{th}$ sublist consists of the $i^{th}$ integer in the argument list repeated i+j times where j is the value of the $i^{th}$ integer. (**Hint:** You may want your function `duplicate_atoms` to use another function `rep` that takes two arguments, `k` and `n`, and returns a list containing `n` repetitions of argument `k`.)

```
(duplicate_atoms '(3 5 3 1)) returns ((3 3 3 3)(5 5 5 5 5 5 5)(3 3 3 3 3 3)(1 1 1 1 1))
```

4. Design and test a Lisp function **infix** that takes as argument a list representing a lisp expression (prefix notation) and returns the equivalent infix version of the expression.

```
(infix '(+ 2 3))      returns      (2 + 3)
(infix '(* (+ 1 2) (- 10 8)))      returns      ((1 + 2) * (10 - 8))
(infix '(* (+ (- 3 4) (/ 4 6)) 5))      returns      (((3 - 4) + (4 / 6)) * 5)
```

You may assume that all operations are binary.

5. Suppose we represented a matrix as a list of lists. For example, `((4 6 8 5)(7 5 2 2)(12 9 6 13)(6 3 3 20))` would represent a 4x4 matrix whose first row contains the elements 4, 6, 8, and 5, and whose second row has the elements 7, 5, 2, and 2, and whose third row has the elements 12, 9, 6, and 13,and whose fourth row has the elements 6, 3, 3, and 20. Write a function `inside_m` that takes a matrix as argument and outputs a list consisting of the elements on the inside of the matrix. The elements should be listed in order from top to bottom and left to right, as shown in the second example below.

```
(inside_m '((4 6 8)(7 5 2)(12 9 6)) )      returns      (5)
(inside_m '((4 6 8 5)(7 5 2 2)(12 9 6 13)(6 3 3 20)) )      returns      (5 2 9 6)
(inside_m '((c a t)(d o g)) )      returns      ()
```

6. Design and test a function `meld_in` that takes as arguments a structure `struct` and a list `lst` and returns the result of melding `struct` into the beginning of each sublist of `lst`.

```
(meld_in '(2 3) '(4 (3 5) 7 (2)))      returns      (4 (2 3 3 5) 7 (2 3 2))
(meld_in '((a b) c) '(7 (5 (6))))      returns      (7 ((a b) c 5 ((a b) c 6)))
(meld_in '(4 a (c)) '((a b) 2 (4 6 (3)) (7) 3 (3 8)) ) returns ((4 a (c) a b) 2 (4 a (c) 4 6 (4 a (c) 3)) (4 a (c) 7) 3 (4 a (c) 3 8))
(meld_in '((7)) '(((a))))      returns      (((7)((7)a)))
```

7. Suppose that a binary tree is represented as a list of the following form

(root left-subtree right-subtree)

An ordered binary subtree is one in which all elements on the left branch of every subtree are smaller than the element in the root of the subtree and all elements on the right branch of every subtree are larger than the element in the root of the subtree. Design and test a function `subtree-leq` that takes as input an integer `int` and an ordered binary tree `tree` of integers and returns a list of those elements in `tree` that are less than or equal to `int`. Note: Do **NOT** first convert the entire tree to a list and then select those elements smaller than `int`. The order of the elements in the result should be from smallest to largest.

```
(subtree_leq 55 '(35 ( 6 (3 () ()) (17 () (19 () ()))) (48 (40 () ()) (74 (60 () ()) ()) )))   returns (3 6 17 19 35 40 48)
(subtree_leq 48 '(35 ( 6 (3 () ()) (17 () (19 () ()))) (72 (41 () ()) ())))    returns     (3 6 17 19 35 41)
(subtree_leq 18 '(35 ( 6 (3 () ()) (17 () (19 () ()))) (72 (41 () ()) ())))    returns     (3 6 17)
(subtree_leq 60 '(42 () (60 (55 () (58 () ())) (80 (73 () ()) (90 () () )))))       returns     (42 55 58 60)
```

8. Suppose that an arithmetic expression is any functional form using only +, *, /, and % (for exponentiation) as operators and using only non-negative numbers, variables (symbols), and (nested) arithmetic expressions as arguments. Two examples are the following:

```
(+ x (* 3 1 4) (* 5 1) 8 (* (* x y z) 0))
(+ x (* 3 1 4) (* 5 1) 8 (* (* (/ x 1) y z) (/ 0 a)))
(+ (* 3 y) (* 4 (% x 5)))
```

Write a function `reduce_exp` which takes a valid arithmetic expression and returns a new one in which the following improvements are made, if they are possible:

- any subexpression consisting of the operator * followed by a list of arguments, one of which is 0, is replaced by 0
- any occurrence of 1 as an argument to * is eliminated, and then, if only one argument remains, the subexpression is replaced by the single argument
- any occurrence of 0 as an argument to + is eliminated, and if only one argument remains, the subexpression is replaced by the single argument
- any subexpression consisting of the operator / whose first argument is 0 is replaced by 0.
- any subexpression consisting of the operator / and two arguments, the second of which is 1, is replaced by the first argument.
- any subexpression consisting of the operator % and two arguments, the second of which is 1, is replaced by the first argument.
- any subexpression consisting of the operator % and two arguments, the second of which is 0, is replaced by the number 1.

Thus `reduce_exp` should produce the following results:

```
(reduce_exp '(+ x (* 3 1 4) (* 5 1) 8 (* (* x y z) 0)))   returns   (+ x (* 3 4) 5 8)
(reduce_exp '(+ x (* 3 1 4) (* 5 1) 8 (* (* (/ x 1) y z) (/ 0 a))))   returns   (+ x (* 3 4) 5 8)
(reduce_exp '(+ (* 3 y) (* 4 (% x 5))))   returns   (+ (* 3 y) (* 4 (% x 5)))
(reduce_exp '(+ (* 0 8) (% x 0)))   returns 1
(reduce_exp '(+ (* 3 y) (* (+ 4 (/ 0 3))(% x 1))))   returns   (+ (* 3 y) (* 4 x))
```

9. We can implement a virtual property list using a technique known as *inheritance*. Suppose we stated that a canary was a bird, and that a bird has two wings. We would like to be able to ask the question "How many wings does a canary have?" and get an answer. One way to do this is to store these facts as properties. For example, we can assert `(setf (get 'bird 'wings) 2)`, meaning that birds have two wings, and `(setf (get 'canary 'isa) 'bird)`, meaning that a canary is a bird. This latter assertion is sometimes called an *isa-link*.

We now need a special function `inherit-get`. `inherit-get` will take two arguments, just like `get`. It will try to do a `get` first, and if this succeeds, will return the value `get` returns. However, if `get` fails, `inherit-get` will use `get` to get the `isa` property of its first argument. If it finds one, `inherit-get` will examine the resulting value to see if it has the desired information. `inherit-get` will keep on going up this *isa-hierarchy* until it finds the property requested or it runs out of isa-links.

For example, suppose we evaluated `(inherit-get 'canary 'wings)`. `inherit-get` will first try a `get` of the `wings` property of `canary`. When this returns `nil`, `inherit-get` then does a `get` of the `wings` property of `bird`, and gets the answer `2`. `inherit-get` returns this as its result. In general, `inherit-get` may go through any number of iterations before it fails or finds an answer.

Write and test a version of `inherit-get` that works as described.

10. Suppose that the hierarchy in problem 9 is a graph instead of a tree — ie., an entity can have more than one isa parent. For example, baking soda isa rising agent, baking soda isa deodorent, and baking soda isa antacid. In this case, the value of the isa property of an entity must be represented as a list; for example, we would assert

```
(setf (get 'baking-soda 'isa) '(rising-agent deodorent antacid))
```

So if we are seeking the *edible* property of baking soda, we first must look and see if baking soda has this property, and if not, then check each of its parents, and then each of their parents, etc. until we either find the property or run out of isa-links. If more than one ancestor has a value for the property that we are seeking, then we select the value assigned to the closest ancestor. (You can arbitrarily choose among them if they are located the same distance away.)

Design and implement a function `full-inherit-get` that works as described.