LING/COMP 445, LING 645
Problem Set 1

There are several types of questions below. For programming questions, please put your answers into a file called `lastname-firstname.clj`. Be careful to follow the instructions exactly and be sure that all of your function definitions use the precise names, number of inputs and input types, and output types as requested in each question.

To do the computational problems, we recommend that you install Clojure on your local machine and write and debug the answers to each problem in a local copy of `lastname-firstname.clj`. You can find information about installing and using Clojure here `https://clojure.org/`.

For questions involving answers in English or mathematics or a combination of the two, put your answers to the question in an **Answer** section like in the example below. You can find more information about LaTeX here `https://www.latex-project.org/`.

Once you have answered the question, please compile your copy of this LaTeXinto a PDF and submit (i) the compiled PDF (ii) the raw file and (iii) your `lastname-firstname.clj` via email to *both* timothy.odonnell@mcgill.ca and savanna.willerton@mail.mcgill.ca. Please make sure all of your code compiles and runs, we cannot give partial credit for code that won't run at all.

Problem 0: This is an example question using some fake math like this $L = \sum_0^\infty \mathcal{G}\delta_x$.

Answer 0: Put your answer right under the question like this $L = \sum_0^\infty \mathcal{G}\delta_x$.

Problem 1: Write a procedure called `abs` that takes in a number, and computes the absolute value of the number. It should do this by finding the square root of the square of the argument. (Note: you should use the `Math/sqrt` procedure built in to Clojure, which returns the square root of a number.)

Answer 1: Answer in `victoria-pittard.clj`.

Problem 2: In both of the following procedure definitions, there are one or more errors of some kind. Explain what's wrong and why, and fix it:

```
(defn take-square
  (* x x))


(defn sum-of-squares [(take-square x) (take-square y)]
  (+ (take-square x) (take-square y)))
```

Answer 2: Fixed functions are in `victoria-pittard.clj`. Here, **take-square** attempts to define a function that takes the square of a number *x* by multiplying it by itself. However, it fails to indicate what the input is (function declarations involve the **defn** keyword, followed by the function name, inputs, and function definition). Next, **sum-of-squares** attempts to add the squares of two numbers, *x* and *y*. While the inputs are indicated in the function declaration, they are not indicated correctly - this function takes each number to the fourth power, as it takes the square of each value as the input. This can be corrected by using *x* and *y* as inputs, rather than the square of each.

Problem 3: The expression `(+ 11 2)` has the value `13`. Write four other different Clojure expressions whose values are also the number `13`. Using `def` name these expressions `exp-13-1`, `exp-13-2`, `exp-13-3`, and `exp-13-4`.

Answer 3: Answer in `victoria-pittard.clj`. Expressions can equate to value 13 in many ways, most simply through addition of various numbers such that they equal 13. For example, 12 + 1, 10 + 3, 9 + 4, and 8 + 5 all equal 13, and can be represented in clojure using **def** and **'** as follows:

```
(def exp-13-1 '(+ 12 1))
(def exp-13-2 '(+ 10 3))
(def exp-13-3 '(+ 9 4))
(def exp-13-4 '(+ 8 5))
```

Problem 4: Write a procedure, called `third`, that selects the third element of a list. For example, given the list `'(4 5 6)`, third should return the number `6`.

Answer 4: Answer in `victoria-pittard.clj`.

Problem 5: Write a procedure, called `compose`, that takes two one-place functions `f` and `g` as arguments. It should return a new function, the composition of its input functions, which computes `f(g(x))` when passed the argument `x`. For example, the function `Math/sqrt` (built in to Clojure from Java) takes the square root of a number, and the function `Math/abs` (also built in to Clojure) takes the absolute value of a number. If we make these functions Clojure native functions using `fn`, then `((compose Math/sqrt Math/abs) -36)` should return `6`, because the square root of the absolute value of `-36` equals `6`.

```
(defn sqrt [x] (Math/sqrt x))
(defn abs [x] (Math/abs x))
((compose sqrt abs) -36)
```

Answer 5: Answer in `victoria-pittard.clj`.

Problem 6: Write a procedure `first-two` that takes a list as its argument, returning a two element list containing the first two elements of the argument. For example, given the list `'(4 5 6)`, `first-two` should return `'(4 5)`.

Answer 6: Answer in `victoria-pittard.clj`.

Problem 7: Write a procedure `remove-second` that takes a list, and returns the same list with the second value removed. For example, given `(list 3 1 4)`, `remove-second` should return `(list 3 4)`

Answer 7: Answer in `victoria-pittard.clj`.

Problem 8: Write a procedure `add-to-end` that takes in two arguments: a list `l` and a value `x`. It should return a new list which is the same as `l`, except that it has `x` as its final element. For example, `(add-to-end (list 5 6 4) 0)` should return `(list 5 6 4 0)`.

Answer 8: Answer in `victoria-pittard.clj`.

Problem 9: Write a procedure, called `reverse`, that takes in a list, and returns the reverse of the list. For example, if it takes in `'(a b c)`, it will output `'(c b a)`.

Answer 9: Answer in `victoria-pittard.clj`.

Problem 10: Write a procedure, called `count-to-1`, that takes a positive integer `n`, and returns a list of the integers counting down from `n` to `1`. For example, given input `3`, it will return `(list 3 2 1)`.

Answer 10: Answer in `victoria-pittard.clj`.

Problem 11: Write a procedure, called `count-to-n`, that takes a positive integer `n`, and returns a list of the integers from `1` to `n`. For example, given input `3`, it will return `(list 1 2 3)`. Hint: Use the procedures `reverse` and `count-to-1` that you wrote in the previous problems.

Answer 11: Answer in `victoria-pittard.clj`.

Problem 12: Write a procedure, called `get-max`, that takes a list of numbers, and returns the maximum value.

Answer 12: Answer in `victoria-pittard.clj`.

Problem 13: Write a procedure, called `greater-than-five?`, that takes a list of numbers, and replaces each number with `true` if the number is greater than `5`, and `false` otherwise. For example, given input `(list 5 4 7)`, it will return `(list false false true)`. Hint: Use the function `map` that we discussed in class.

Answer 13: Answer in `victoria-pittard.clj`.

Problem 14: Write a procedure, called `concat-three`, that takes three sequences (represented as lists), `x`, `y`, and `z`, and returns the concatenation of the three sequences. For example, given the sequences `(list 'a 'b)`, `(list 'b 'c)`, and `(list 'd 'e)`, the procedure should return `(list 'a 'b 'b 'c 'd 'e)`.

Answer 13: Answer in `victoria-pittard.clj`.

Problem 15: Write a procedure, called `sequence-to-power`, that takes a sequence (represented as a list) `x`, and a positive integer `n`, and returns the sequence x^n. For example, given the sequence `(list 'a 'b)` and the number `3`, the procedure should return `(list 'a 'b 'a 'b 'a 'b)`.

Answer 15: Answer in `victoria-pittard.clj`.

Problem 16: Define L as a language containing a single sequence, $L = a$. Write a procedure `in-L?` that takes a sequence (represented as a list), and decides if it is a member of the language L^*. That is, given a sequence x, the procedure should return `true` if and only if x is a member of L^*, and `false` otherwise.

Answer 16: Answer in `victoria-pittard.clj`.

Problem 17: Let A and B be languages. We'll use $\textsc{concat}(A, B)$ to denote the concatenation of A and B, in that order. Find an example of languages A and B such that $\textsc{concat}(A, B) = \textsc{concat}(B, A)$.

Answer 17: The concatenation of two languages A and B (in that order) is equivalent to the set of the possible concatenations of a morpheme from B to the end of a morpheme from A. For example, if language A includes *{a, b, c}* and language B includes *{d, e}*, then the concatenation of A and B includes *{ad, ae, bd, be, cd, ce}*. As such, the concatenation of A and B equals the elements of A and B are identical, or in the event of an empty language A or B, as these are the only ways to guarantee each element behaves as both a suffix and prefix in each scenario, and thus identical roles when the opposite concatenation occurs (as is the case when A equals B) or that the concatenation yields an empty set (as is the case when either A or B is empty). For example, if A = *{a, b}* and B = *{a, b}*, both the concatenation of A and B and the concatenation of B and A will result in *{aa, ab, ba, bb}*.

Problem 18: Let A and B be languages. Find an example of languages A and B such that $\textsc{concat}(A, B)$ does not equal $\textsc{concat}(B, A)$

Answer 18: Following the same logic as shown above in **Answer 17**, an example of two languages such that the concatenation of A and B does not equal the concatenation of B and A is any case where A and B do not have identical elements and neither is an empty language. For example, if A = *{a, b}*, and B = *{c, d}*, then the concatenation of A and B is *{ac, ad, bc, bd}*, while the concatenation of B and A is *{ca, cb, da, db}*. These two concatenations are not equivalent.

Problem 19: Find an example of a language L such that $L = L^2$, i.e. $L = \textsc{concat}(L, L)$.

Answer 19: Again following the logic shown in **Answer 17** regarding how concatenation of two languages work, there are some languages L that are equivalent to the concatenation of L to itself. An example would be the empty language, L = *{}*, as the concatenation of two empty elements (with each element in this case being a language) is yet another empty element (empty language), thus identical to the initial elements (languages).

Problem 20: Argue that the intersection of two languages L and L' is always contained in L.

Answer 20: In set theory, the intersection of two sets A and B is the set of all elements that exist in both A and B (effectively, the overlap). As a language is a set of grammatical strings, the same concept applies, and the intersection of L and L' is the set of all elements that exist in both L and L'. As the elements must exist in **both** L and L', it is trivial to conclude that each element must exist in L and L' as individual languages ass well, thus proving that the intersection of L and L' is always contained in L.

Problem 21: Let L_1, L_2, L_3, and L_4 be languages. Argue that the union of Cartesian products $(L_1 \times L_3) \cup (L_2 \times L_4)$ is always contained in the Cartesian product of unions $(L_1 \cup L_2) \times (L_3 \cup L_4)$.

Answer 21:

To show that $(L_1 \times L_3) \cup (L_2 \times L_4)$ is always contained in $(L_1 \cup L_2) \times (L_3 \cup L_4)$, we must first consider the individual operations used. The union of two sets (with each set in this case being the Cartesian product of two languages) A and B is defined as the set composed of all elements existing in either A, B, or both A and B (in other words, existing in at least one of A and B). The Cartesian product of two sets (in this case, each set is a language) A and B is defined as the set of all possible ordered pairs of elements in A followed by elements in B. In the case of a language, these ordered pairs refer to a concatenation. Thus, the Cartesian product of two languages is the result of concatenating every string in the first language with every string in the second language.

Given these definitions, we know that the Cartesian product of L1 and L3 is the set of all strings in L1 followed by all strings in L3 and the Cartesian product of L2 and L4 is the set of all strings in L2 followed by all strings in L4. Thus, the union of these two Cartesian products is the set of all strings in L1 followed by all strings in L3, and all strings in L2 followed by all strings in L4.

From the same definitions, we know that the union of L1 and L2 is the set of all strings in at least one of L1 and L2 (effectively, the strings in L1 and the strings in L2, removing possible overlap). Using the same logic, the union of L3 and L4 is the set of all strings in at least one of L3 and L4 (effectively, the strings in L3 and L4, removing possible overlap. Thus, the Cartesian product of these two unions is the set of all strings in either L1 or L2 followed by all strings in either L3 or L4. Logically, it is trivial to see that this is all strings in L1 followed by L3, all strings in L1 followed by L4, all strings in L2 followed by L3, and all strings in L2 followed by L4 by rephrasing the definition of a union.

Given this, it is clear that $(L_1 \times L_3) \cup (L_2 \times L_4)$ is always contained in $(L_1 \cup L_2) \times (L_3 \cup L_4)$ as the set of all strings made by concatenating **L1 and L3** and all strings made by concatenating **L2 and L4** is trivially included in the set of all strings made by concatenating **L1 and L3**, L1 and L4, L2 and L3, and **L2 and L4** (removing repeats).

Problem 22: Let L and L' be finite languages. Show that the number of elements in the Cartesian product $L \times L'$ is always equal to the number of elements in $L' \times L$.

Answer 22: The concatenation of two languages is the set of all strings created by the concatenation of each string in one language (L) and each string in the other (L'). For example, if L = *{a, b, c}* and L' = *{d, e, f}*, the concatenation of L and L' is *{ad, ae, af, bd, be, bf, cd, ce, cf}* and the concatenation of L' and L is *{da, db, dc, ea, eb, ec, fa, fb, fc}*. In this example, it is clear that the cardinality of L CONCAT L' and the cardinality of L' CONCAT L are equal, as each element in one is the reverse of an element in the other, with no additional elements existing. The same holds true for all finite languages, as shown by the more rigorous proof below:

Proof. As all languages can be represented as sets, to show that |LxL'| = |L'xL| it is sufficient to show that the cardinality of the cartesian product of two sets, A and B, is commutative. First, if either A or B is empty, then trivially the cartesian product of the two must be empty as well (as there is either nothing to concatenate to strings in A, or no strings to concatenate the strings in B to). In other words, if either A or B has a cardinality of zero, the cartesian product AxB has a cardinality of zero. Thus, */AxB/ = /A/ x /B/* in the base case where either A or B is empty. Otherwise, both |A| and |B| are greater than zero, and can be represented as x and y, respectively. Thus, the elements a_i in A can be represented as $A = \{a_1, a_2, ..., a_x\}$, and the elements b_i in B can be represented as $B = \{b_1, b_2, ..., b_y\}$. As the cartesian product of A and B is the set of ordered pairs made by concatenating b_i onto a_i for all values of i, A x B

$= \{a_1b_1, a_1b_2, ..., a_1b_y, a_2b_{1::y}, ..., a_xb_{1::y}\}$. This makes it quite clear to see that for each of the x elements in A, there are y elements in A x B corresponding to the concatenation of each of the y elements of B to this element of A. From this, it is clear that |A x B| = |A| x |B|. As the cardinality of a set is always an integer, in both this case and in the base case where one of the sets is empty, this can be further reduced to |A x B| = $x * y$, where x and y are the cardinalities of A and B, respectively. As integer multiplication follows the commutative property, we can thus conclude that the cartesian product of two sets, and therefore of two languages, also follows the commutative property. □

Problem 23: Suppose that the concatenation of a language L is equal to itself: `concat`$(L, L) = L$. Show that L is either the empty set or an infinite language. (Remember that the empty set contains the null string.)

Answer 23: Let L = *{a}* serve as our base case where L is not empty (effective, because it has one element). The concatenation of L to itself here yields *{aa}*, which is not equivalent to *{a}*. Given a's role as a placeholder here, it is sufficient to say that a language containing one string cannot be equivalent to the cartesian product of it with itself. The only way for *aa* to also exist in L, not just L CONCAT L, is if it also has the elements of L concatenated to it, causing *aaa* and *aaaa* to also exist in L CONCAT L. The only way for *aaa* and *aaaa* to also exist in L (required for equivalency) is if it also has the elements of L concatenated to it, causing more elements *aaaa*, *aaaaa*, *aaaaaa*, *aaaaaaa*, and *aaaaaaaa* to also exist in L. This pattern continues, and the only way all of this exponentially increasing number of elements in both L and L CONCAT L is if the set is infinite, as no base case is reached by increasing the number of elements in both L and L CONCAT L. Thus, the language L must either be infinite or the empty language (as shown in **Answer 19**).