**CSSE 304 Final Exam Part 2 (non-interpreter part)** Monday evening, Feb 24, 2020 **Name\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

You may not use code written by other CSSE 304 students (from this term or any previous term), or code that you find on the internet.

You may receive some partial credit based on your code and comments. If your code passes the test cases but does not implement the required features according to the specification, you may receive less than full credit.

**Cautions!** Do not get so caught up in getting all of the points for one problem that you do not have time to work on the other one.

Do not write code that is specific to my test-cases but does not solve the general problem.

**There are three different assignments on the PLC server; one for each problem**

Please put your solutions in three different files.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Problem** | **Max score** | **Your score** | **Instructor/grader comments** |
| same-fringes | 1 | 10 |  |  |
| stack enhancement | 2 | 10 |  |  |
| imperative form | 3 | 10 |  |  |
|  | **Total** | **30** |  |  |

1. (12 points) The **same-fringes-start.ss** file contains the corrected same-fringe code from class. You are to write the same-fringes? procedure. It takes any number of snlists (with the same modified definition as in the Day 37 slides) as arguments. Returns #t if they all have the same fringe, and #f otherwise. **Requirements:**
   1. Must short-circuit like the example from class. So if the first leaves from all of the snlists are not identical, it returns #f without looking at any of the second leaves.
   2. Must create a sf-coroutine for each of its arguments, and it must have communication between the driver and each sf-coroutine that is similar to the given code.

There is probably no need for you to modify make-sf-coroutine.

> (same-fringes? '(1) '(1) '(1))

#t

> (same-fringes? '(1) '(2) '(1))

#f

> (same-fringes? '(1 (2)) '(() 1 2) '((1) 2) '(1 () ((2))))

#t

> (same-fringes? '(1 (2)) '(() 1 2) '((1) 2 (3)) '(1 () ((2))))

#f

> (same-fringes? '((1)))

#t

> (same-fringes? '((1)) '(1 ()) '(() 1))

#t

> (same-fringes? '(1 2) '((1 2)) '(1 (((2)))) '((((1)) ((2)))) '((1 () (() 2))))

#t

> (same-fringes? '(1 2) '((1 2)) '(1 (((2)))) '((((1)) ((2) 3))) '((1 () (() 2))))

#f

Submit your code for this problem to the **F202020-p1** assignment on the PLC server.

1. (10 points) Recall the stack “object” implementation that we used for A9. (You can find the original code in the A9 homework folder, or, better, in your A9 code). Add the following methods.

* size: computes **in constant** time and returns the current number of items in this stack.
* top: returns the current item that is on top of the stack. Does not remove the item.
* push-count: returns the total number of times that the push method has been called for all stacks.

Your code does not have to check for input errors such as trying to pop an empty stack.

**Examples:**

> **(let ([s (make-stack)])**

**(s 'push 1)**

**(s 'push 2)**

**(s 'pop)**

**(s 'push 3)**

**(s 'pop)**

**(list (s 'top) (s 'size) (s 'push-count)))**

(1 1 3)

> **(let ([s1 (make-stack)]**

**[s2 (make-stack)]**

**[s2-top #f])**

**(s1 'push 1)**

**(s1 'push 2)**

**(s2 'push 3)**

**(set! s2-top (s2 'top))**

**(s2 'push (s1 'pop))**

**(s2 'push (s1 'pop))**

**(s2 'pop)**

**(s1 'push 5)**

**(s1 'push (s2 'top))**

**(s1 'pop)**

**(list (s1 'top)**

**(s1 'size)**

**(s2 'top)**

**(s2 'size)**

**(s1 'push-count)**

**(s2 'push-count)**

**s2-top))**

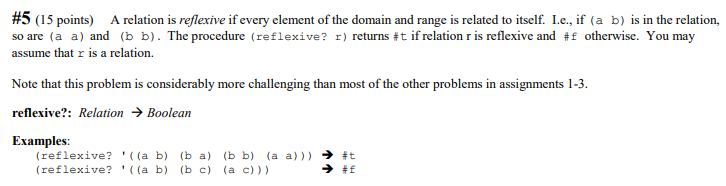
(5 1 2 2 10 10 3)

Note that the push-counts above represent the situation where make-stack is defined, and then these two test cases are run in the order given.

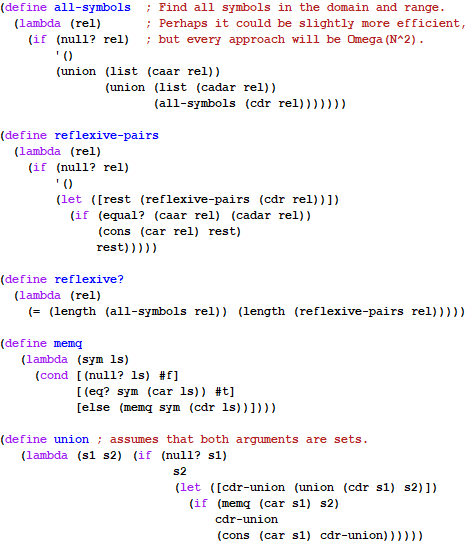
Submit your code for this problem to the **F202020-p2** assignment on the PLC server.

If you finish these non-interpreter problems before you finish the interpreter computer part, please go ahead and turn in this paper, so that we can begin grading your work.

1. *Imperative form:* Assignment 3 contained the following problem:



This is one of my solutions. It counts the number of different symbols in the relation, counts the number of reflexive pairs, and checks to see whether they are the same. It assumes that both relations are *sets* of ordered pairs of symbols.



The **imperative-start-code.ss** file contains this code, a cps-with-datatype-continuations version, and a significant amount of code for an imperative-form version. All of the places where you need to add code are marked with 'fill-it-in.   
To assist you with debugging, I improved my trace-it code from class, so that it only shows the global variables that are actually used by the particular procedure that is being traced. You can turn tracing on and off by setting the value of the \*tracing\* variable.

Submit your code for this problem to the **F202020-p3** assignment on the PLC server.