Homework Assignment 4

Any automatically graded answer may be manually graded by the instructor. Submissions are expected to only use functions taught in the course. If a submission uses a disallowed function, that exercise can get zero points. *All functions that mutate values are disallowed* (mutable functions usually have a `!` in their name).

Infinite Streams

1. Implement the notion of accumulator for infinite streams.[1] Given a stream `s` defined as

   ```
   e0   e1   e2  ...
   ```

   Function `(stream-fold f a s)`

   ```
     a   (f e0 a)   (f e1 (f e0 a))   (f e2 (f e1 (f e0 a)))   ...
   ```

2. Implement a function that advances an infinite stream a given number of steps. Given a stream `s` defined as

   ```
   e0   e1   e2   e3   e4   e5   ...
   ```

   Function `(stream-skip 3 s)`

   ```
   e3   e4   e5  ...
   ```

Finite streams as sets

We will define `set` as a finite stream. Finite streams can be used to represent potentially infinite data structures.

The goal of this exercise is to develop a library of *regular expression generators*. In this assignment *a promise-list represents a set* and we explore regular expressions as a technique to generate possibly infinite sets (promise-lists). Regular expressions are also discussed in the context of regular languages in (CS420) and as the basis of lexing in compilers (CS451).

3. Define `set-void` that represents $\emptyset$, the empty set (the empty promise list).
4. Define `set-epsilon` that represents $\{\epsilon\}$, the set containing only the empty string (formally ϵ, `""` in Racket).
5. Define `(set-char c)` that represents $\{c\}$, a set that contains a string with a single character `c`. In Racket, a character, say `#\a`, can be converted into a string with function `string`. See the manual page[2] on characters to learn more.
6. Define function `(set-prefix u p)` that prepends string `u` on every element of a promise list `s`. We can specify the prefix function as $prefix(u, s) = \{u \cdot v \mid v \in s\}$, where $u \cdot v$ is string concatenation (`string-append` in Racket).

[1] Recall that `fold` is the accumulator for lists and was taught in class.

[2] `https://docs.racket-lang.org/guide/characters.html`

7. Define `set-union` that represents the set union $\cup$. The implementation of `set-union` *must* interleave each element of `p1` with an element of `p2` (following the same requirements of function `interleave` of Homework 3). Interleaving is desirable because if `p1` is infinite and we simply concatenate the two promise lists, then we would never observe elements of `p2`.

8. Define function `set-concat` represents set concatenation $\circ$, which concatenates every pair of strings from both sets. We can specify set concatenation as $p_1 \circ p_2 = \{u \cdot v \mid u \in p_1 \wedge v \in p_2\}$. Alternatively, we give an inductive specification:

$$\begin{aligned} \emptyset \circ p_2 &= \emptyset \\ p_1 \circ p_2 &= prefix(u, p_2) \cup \left(p_1' \circ p_2\right) && \text{if } p_1 = \{u\} \cup p_1' \end{aligned}$$

Handling expressions

9. Extend functions `+` and `*` to support multiple-arguments (including zero arguments).

10. Implement a function that outputs a string-representation of the given expression.