Homework Assignment 3

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. Excluding promises, *all functions that mutate values are disallowed* (mutable functions usually have a `!` in their name).

Promise lists as sets

Let a *promise list* be either: a promise holding `empty`, or a promise holding a pair whose right element is a promise list. Promise lists can be used to represent potentially infinite data structures. The accompanying test file includes a short tutorial and utility functions on the promise-list data strucutre.

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).

1. Define `p:void` that represents $\emptyset$, the empty set (the empty promise list).

2. Define `p:epsilon` that represents $\{\epsilon\}$, the set containing only the empty string (formally ϵ, `""` in Racket).

3. Define `(p: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[1] on characters to learn more.

4. Define `p:union` that represents the set union $\cup$. The implementation of `(p:union p1 p2)` *must* interleave each element of `p1` with an element of `p2` (see test cases). Interleaving is desirable because if `p1` is infinite and we simply concatenate the two promise lists, then we would never observe elements of `p2`.

5. Define function `(p: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).

6. Define function `p:cat` 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:

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

7. Implement function `(p:star union pow p)` that takes a union operator as parameter, a power operator as parameter, and a set (promise list) `p`.

$$p^{\star} = p^0 \cup p^1 \cup \cdots \cup p^n \cup p^{n+1} \cup \cdots$$

   Where p^n is `(pow p n)`. The reason we take `union` and `pow` as parameters is so that we can grade your solution even if you don't implement `union`. You can find examples of the Kleene Star in Wikipedia.[2]

[1] `https://docs.racket-lang.org/guide/characters.html`
[2] `https://en.wikipedia.org/wiki/Kleene_star`

NOTES

- Function `(promise? p)` returns `#t` if, and only if value `p` is a promise, otherwise it returns `#f`.
- Careful when comparing promises. `(equal? (delay 1) (delay 1))` returns `#f` as it compares the promise's reference, not its contents.

Infinite Streams

8. Implement the notion of accumulator for infinite streams.[3] Given a stream `s` defined as

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

   Function `(stream-foldl f a s)`

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

9. 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  ...
   ```

Evaluating expressions

10. Extend functions `r:eval-exp` with support for booleans.
    (a) Implement a data structure `r:bool` (using a `struct`) with a single field called `value` that holds a boolean.
    (b) Extend the evaluation function to support boolean values.
    (c) Extend the evaluation to support binary-operation `and`. The semantics of `and` **must** match Racket's operator `and`. Recall that Racket's `and` is not a variable to a function, but a special construct, so its usage differs from function `+`, for instance.
    (d) Extend function `+` to support multiple-arguments (including zero arguments).
    (e) Extend primitive `and` to support multiple-arguments (including zero arguments).

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