1. Evaluation: Adding a new form

We would like to introduce a new special form to our evaluator called `same?`. `Same?` always takes three arguments, and returns `#t` if all three arguments are `eq?`. `Same?`, however is smart in that if the first two arguments are different, then the third argument is not evaluated. Here are some examples of using `same?`.

\[
\begin{align*}
(same? 'x 'x 'x) & \quad \Rightarrow \quad #t \\
(same? 'x 'x 'y) & \quad \Rightarrow \quad #f \\
(same? 'x 'y 'y) & \quad \Rightarrow \quad #f \\
(same? 'x 'y (/ 1 0)) & \quad \Rightarrow \quad #f \\
(same? 'x 'x (/ 1 0)) & \quad \Rightarrow \quad \text{Divide by Zero Error}
\end{align*}
\]

To add this special form to the evaluator, we need to define some data abstraction.

Define the function `same??` that checks to see if an expression is a `same?` expression.

\[
\begin{align*}
\text{(define (same?? exp) (tagged-list? exp 'same))}
\end{align*}
\]

Define the functions `same?-first` and `same?-second` that select out the first and second sub-expressions (assume someone else defined `same?-third`).

\[
\begin{align*}
\text{(define (same?-first exp) (cadr exp))} \\
\text{(define (same?-second exp) (caddr exp))}
\end{align*}
\]

Next, write the appropriate clause to add to the `cond` clause of `eval`, assuming that we have the function `eval-same?` that will evaluate a `same?` expression.

\[
\begin{align*}
\text{((same?? exp) (eval-same? exp env))}
\end{align*}
\]

Finally, write the `eval-same?` function that takes a `same?` expression and an environment and implements the special form as described above.

\[
\begin{align*}
\text{(define (eval-same? exp env)} \\
\text{ (let ((x (eval (same?-first exp) env)))} \\
\text{ (if (eq? x (eval (same?-second exp env)))} \\
\text{ (eq? x (eval (same?-third exp) env)))))}
\end{align*}
\]

We added `same?` as a special form in our language because our language had applicative order evaluation. If, instead, our language had normal order evaluation, then we could simply define `same?` as a function.
Assuming our Scheme has the normal order evaluation, define \texttt{same?} as a function.

\begin{verbatim}
(define (same? a b c)
  (if (eq? a b)
      (eq? a c)
      #f))
\end{verbatim}

Now implement \texttt{eval-same?} by doing a syntactic transformation to an \texttt{if} expression (that also uses \texttt{eq?}). Assume \texttt{eq?} is implemented in the evaluator.

\begin{verbatim}
((same?? exp) (eval (same?->if exp) env))
(define (same?->if exp)
  (list 'if (list 'eq? (same?-first exp) (same?-second exp))
        (list 'eq? (same?-first exp) (same?-third exp))
        #f))
\end{verbatim}

2. Streams

Create a stream of Fibonacci numbers using \texttt{add-streams}.

\begin{verbatim}
(define fibs
  (cons-stream 1
              (cons-stream 1
                            (add-stream fibs (stream-cdr fibs)))))
\end{verbatim}

Create a stream of powers of \textit{x}. (Hint: write a procedure called \texttt{scale-stream}).

Eg for \textit{x} = 5:

\begin{verbatim}
(define (scale-stream a k)
  (cons-stream (* (stream-car a) k)
               (scale-stream (stream-cdr a) k)))

(define (powers x)
  (cons-stream 1
               (scale-stream (powers x) x)))
\end{verbatim}

Create a stream of factorials. (Hint: write a procedure called \texttt{mult-streams} that multiplies 2 streams).

\begin{verbatim}
(define (mult-streams a b)
  (cons-stream (* (stream-car a) (stream-car b))
               (scale-stream (stream-cdr a) (stream-cdr b))))
\end{verbatim}
Write an approximation to \( e^x \) by creating a series in which every successive element is a better approximation to \( e^x \). Remember that 
\[
    e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}
\]  
(Hint: write \texttt{stream-acc}um \texttt{and} \texttt{div-streams}). To get 
\[
    \sum_{n=0}^{20} \frac{x^n}{n!}
\]  
for example, you would do \texttt{(stream-ref result 20)}.

```scheme
(define (stream-accum s)
    (cons-stream (stream-car s)
                 (add-streams (stream-cdr s)
                              (stream-accum s)))))

(define (div-streams a b)
    (cons-stream (/ (stream-car a) (stream-car b))
                 (div-streams (stream-cdr a) (stream-cdr b)))))

(define (e x)
    (stream-accum (stream-div (powers x) facts)))
```