Streams

1. Streams WWSD

    scm> (define a (cons-stream 4 (cons-stream 6 (cons-stream 8 a))))
    scm> (car a)
        4
    scm> (cdr a)
        #[promise (not forced)]
    scm> (cdr-stream a)
        (6 . #[promise (not forced)])
    scm> (define b (cons-stream 10 a))
    scm> (cdr b)
        #[promise (not forced)]
    scm> (cdr-stream b)
        (4 . #[promise (forced)])
    scm> (define c (cons-stream 3 (cons-stream 6 a)))
    scm> (cdr-stream c)
    Error: too few operands in form

What elements of a, b, and c have been evaluated thus far?
2. Write a function `merge` that takes in two sorted infinite streams and returns a new infinite stream containing all the elements from both streams, in sorted order.

```
(define (merge s1 s2)
  (if (null? s1)
      s2
      (if (<= (car s1) (car s2))
          (cons-stream (car s1) (merge (cdr-stream s1) s2))
          (cons-stream (car s2) (merge s1 (cdr-stream s2))))
  )
)

; Alternate solution
(define (merge s1 s2)
  (if (null? s1)
      s2
      (if (<= (car s1) (car s2))
          (cons-stream (car s1) (merge (cdr-stream s1) s2))
          (merge s2 s1)
      )
  )
)
```

3. Write a function `half_twos_factorial` that returns a new stream containing all of the factorials that contain the digit 2 divided by two. Your solution must use only the following functions, without defining any additional ones. Likewise, any lambda expressions should contain only calls to the following functions or built in functions.

```
; Returns a new Stream where each new value is the result of calling fn on the value in the stream s
```
(define (map-stream s fn)
  (if (null? s) s
      (cons-stream (fn (car s)) (map-stream (cdr-stream s) fn)))))

; Returns a new Stream containing all values in the stream s that
; satisfy the predicate fn
(define (filter-stream s fn)
  (cond ((null? s) s)
        ((fn (car s)) (cons-stream (car s) (filter-stream (cdr-stream s) fn)))
        (else (filter-stream (cdr-stream s) fn)))))

; Returns True if n contains the digit 2. False otherwise
(define (contains-two n)
  (cond ((= n 0) #f)
        ((= (remainder n 10) 2) #t)
        (else (contains-two (quotient n 10)))))

; Returns the factorial n
(define (factorial n)
  (if (= n 0) 1 (* n (factorial (- n 1)))))

; Returns a stream of factorials
(define (factorial-stream)
  (define (helper n)
    (cons-stream (factorial n) (helper (+ n 1))))
  (helper 1))

Fill in the skeleton below.

(define (half-twos-factorial)
  (map-stream (filter-stream (factorial-stream) contains-two)
               (lambda (x) (quotient x 2))))

Tail Recursion

1. For the following procedures, determine whether or not they are tail recursive. If they are not, write why they aren’t and rewrite the function to be tail recursive to the right.

; Multiplies x by y
(define (mult x y)
  (if (= 0 y)
      0
      (+ x (mult x (- y 1)))))

(define (mult x y)
  (define (helper x y total)
    (if (= 0 y)
        total
        (helper x (- y 1) (+ total x)))
  (helper x y 0))

Notn tail recursive—after evaluating the recursive call, we still need to apply ‘+’, so evaluating the recursive call is not the last thing we do in the frame. (Review non-tail recursive factorial, the reason that is not tail recursive applies to this procedure)

; Always evaluates to true
; assume n is positive
(define (true1 n)
  (if (= n 0)
      #t
      (and #t (true1 (- n 1)))))

Tail recursive—the recursive call to “true1” is the final sub-expression of the `and` special form. Therefore, we will not need to perform any additional work after getting the result of the recursive call.

; Always evaluates to true
; assume n is positive
(define (true2 n)
  (if (= n 0)
      #t
      (or (true2 (- n 1)) #f))

(define (true2 n)
  (if (= n 0)
      #t
      (true2 (- n 1)))))
Not tail recursive—the recursive call to “true2” is not the final sub-expression of the `or` special form. Even though it will always evaluate to `true` and short-circuit, the interpreter does not take that into account when determining whether to evaluate it in a tail context or not.

; Returns true if x is in lst
(define (contains lst x)
  (cond ((null? lst) #f)
        ((equal? (car lst) x) #t)
        ((contains (cdr lst) x) #t)
        (else #f)))

Not tail recursive—the recursive call to “contains” is in a predicate sub-expression. That means we will have to evaluate another expression if it evaluates to true, so it is not the final thing we evaluate.

(define (contains lst x)
  (cond ((null? lst) #f)
        ((equal? (car lst) x) #t)
        (else (contains (cdr lst) x))))

2. Rewrite this function tail-recursively.

; Returns a list of pairs, the ith pair has item as its car and the ; ith element of lst as its cdr
(define (add-to-all item lst)
  (if (null? lst)
      lst
      (cons (cons item (car lst))
            (add-to-all item (cdr lst)))))

(define (add-to-all item lst)
  (define (helper item lst so-far)
    (if (null? lst)
        so-far
        (helper item (cdr lst) (append so-far (list (cons item (car lst)))))))
  (helper item lst nil))
3. Implement \texttt{sum-satisfied-k} which, given an input list \texttt{lst}, a predicate procedure \texttt{f} which takes in one argument, and an integer \texttt{k}, will return the sum of the first \texttt{k} elements that satisfy \texttt{f}. If there are not \texttt{k} such elements, return 0.

\begin{verbatim}
; Doctests
scm> (define lst `(1 2 3 4 5 6))
scm> (sum-satisfied-k lst even? 2)  ; 2 + 4
6
scm> (sum-satisfied-k lst (lambda (x) (= 0 (modulo x 3))) 10)
0
scm> (sum-satisfied-k lst (lambda (x) #t) 0)
0
\end{verbatim}

Implement \texttt{sum-satisfied-k} tail recursively.

\begin{verbatim}
(define (sum-satisfied-k lst f k)
  (define (sum-helper lst k total)
    (cond ((= 0 k) total)
          ((null? lst) 0)
          ((f (car lst)) (sum-helper (cdr lst) (- k 1)
                        (+ total (car lst))))
          (else (sum-helper (cdr lst) k total))))
  (sum-helper lst k 0))
\end{verbatim}

4. Implement \texttt{remove-range} which, given one input list \texttt{lst}, and two nonnegative integers \texttt{i} and \texttt{j}, returns a new list containing the elements of \texttt{lst} in order, without the elements from index \texttt{i} to index \texttt{j} inclusive. For example, given the list \((0 \ 1 \ 2 \ 3 \ 4)\), with \texttt{i} = 1 and \texttt{j} = 3, we would return the list \((0 \ 4)\). You may assume \texttt{j} \textgreater{} \texttt{i}, and \texttt{j} is less than the length of the list. (Hint: you may want to use the built-in append function, which returns the result of appending the items of all lists in order into a single well-formed list.)

\begin{verbatim}
; Doctests
scm> (remove-range `(0 1 2 3 4) 1 3)
(0 4)

(define (remove-range lst i j)
  (define (helper lst index)
    (cond ((> index j) lst)
          ((>= index i) (helper (cdr lst) (+ index 1)))
          (else (cons (car lst) (helper (cdr lst) (+ index 1))))))
  (helper lst 0))
\end{verbatim}
Now implement \texttt{remove-range} tail recursively.

\begin{verbatim}
(define (remove-range lst i j)
  (define (remove-tail lst index so-far)
    (cond ((> index j) (append so-far lst))
      ((>= index i)
       (remove-tail (cdr lst) (+ index 1) so-far))
      (else (remove-tail
              (cdr lst)
              (+ index 1)
              (append so-far (list (car lst))))))
  (remove-tail lst 0 nil)))
\end{verbatim}

Interpreters

1. For the following questions, circle the number of calls to scheme\_eval and the number of calls to scheme\_apply:

\begin{verbatim}
scm> (+ 1 2)
3
\end{verbatim}

\begin{tabular}{|c|c|c|c|}
\hline
\textbf{Calls to scheme\_eval:} & 1 & 3 & 4 & 6 \\
\hline
\end{tabular}
Calls to scheme_apply: | 1 | 2 | 3 | 4

1. **Evaluate** entire expression
   a. **Evaluate** +
   b. **Evaluate** 1
   c. **Evaluate** 2
   d. **Apply** + to 1 and 2

```
scm> (if 1 (+ 2 3) (/ 1 0))
5
```

Calls to scheme_eval: | 1 | 3 | 4 | 6

Calls to scheme_apply: | 1 | 2 | 3 | 4

1. **Evaluate** entire expression
   a. **Evaluate** predicate 1
   b. Since 1 is true, **evaluate** <if-true> sub-expression
      i. **Evaluate** +
      ii. **Evaluate** 2
      iii. **Evaluate** 3
      iv. **Apply** + to 2 and 3
   
Note that we never needed to evaluate the **if** because it's one of our special forms!

```
scm> (or #f (and (+ 1 2) 'apple) (- 5 2))
apple
```

Calls to scheme_eval: | 6 | 8 | 9 | 10

Calls to scheme_apply: | 1 | 2 | 3 | 4

1. **Evaluate** entire expression
   a. **Evaluate** first sub-expression of or
   b. **Evaluate** second sub-expression of or
      i. **Evaluate** first sub-expression of and
1. **Evaluate** +
2. **Evaluate** 1
3. **Evaluate** 2
4. **Apply** + to 1 and 2
   ii. **Evaluate** 'apple
   iii. Since the and expression evaluates to true, we short circuit

Note that we never needed to evaluate the or or the and because they are special forms!

scm> (define (add x y) (+ x y))
add
scm> (add (- 5 3) (or 0 2))
2

<table>
<thead>
<tr>
<th>Calls to scheme_eval:</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calls to scheme_apply:</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

1. **Evaluate** entire define expression
2. **Evaluate** call to add function
   a. **Evaluate** add function
   b. **Evaluate** first argument
      i. **Evaluate** -
      ii. **Evaluate** 5
      iii. **Evaluate** 3
      iv. **Apply** - to 5 and 3
   c. **Evaluate** second argument (the or expression)
      i. **Evaluate** 0
      ii. Since 0 is #t, we short circuit
   d. **Apply** add function to 2 and 0 (enter body of the function)
      i. **Evaluate** the body of the function
         1. **Evaluate** +
         2. **Evaluate** x to be 2
         3. **Evaluate** y to be 0
         4. **Apply** + to 2 and 0

---

**Macros**

**Question 0**
What will Scheme output? If you think it errors, write Error.

```scheme
scm> (define-macro (doierror) (/ 1 0))
doierror
scm>(doierror)
Error
scm> (define x 5)
x
```
Question 1
Consider a new special form, when, that has the following structure:
(when <condition> <expr1> <expr2> <expr3> ...)
If the condition is not false (a truthy expression), all the subsequent operands are evaluated in order and the value of the last expression is returned. Otherwise, the entire when expression evaluates to okay.

```
scm> (when (= 1 0) (/ 1 0) 'error)
okay
scm> (when (= 1 1) (print 6) (print 1) 'a)
6
1
a
```

Create this new special form using a macro. Recall that putting a dot before the last formal parameter allows you to pass any number of arguments to a procedure, a list of which will be bound to the parameter, similar to (*args) in Python.

a) Fill in the skeleton below to implement this without using quasiquotes.
```
(define-macro (when condition . exprs)
  (list 'if condition (cons 'begin exprs) ''okay))
```

b) Now, implement the macro using quasiquotes.
```
(define-macro (when condition . exprs)
  `(if ,condition ,(cons 'begin exprs) 'okay))
```

Question 2
Define a macro while that processes a while loop by converting it to a tail recursive function.
The goal of this question is to define a macro that represents a while loop. Since this is a difficult task we will break it into parts.

2a
Write tail-recursive factorial:
```
(define (fact n)
  (define (fact-tail n result)
    (if (= n 0)
      result
      (fact-tail (- n 1) (* n result)))
  )
```
Using the above problem to assist implementation, create the while macro. This macro will accept 4 arguments:

- initial-bindings: this will represent initialization values for variables in the loop
- condition: this will represent the condition which the while loop should continue to check to see if the loop should continue
- return: after the loop has ended this represents the value that should be returned

You may find the built-in map function useful for this problem:

```
scm > (map (lambda (x) (* 2 x)) '(1 2 3))
(2 4 6)
```

And here’s an example of the while macro being used to calculate the factorial:

```
scm > (define (fact n)
    (while
        ((acc 1) (n n))
        (> n 0)
        ((* acc n) (- n 1))
        acc))

fact
scm> (fact 4)
24
```

Fill in the following macro definition:

```
(define (cadr lst) (car (cdr lst)))

(define-macro (while initial-bindings condition updates return)
    (define helper-vars (map car initial-bindings))
    (define initial-vals (map cadr initial-bindings))
    (list 'begin
        (list 'define (cons 'helper helper-vars)
            '(if ,condition
                ,(cons 'helper updates)
                ,return))
        (cons 'helper initial-vals)))
```
CONGRATULATIONS!
You made it to the end of the worksheet! Great work :)
