0.1 Warm Up: Conditional Expressions

1. What does Scheme print?

s

cm> (if (or #t (/ 1 0)) 1 (/ 1 0))

s

cm> (if (> 4 3) (+ 1 2 3 4) (+ 3 4 (* 3 2)))

s

cm> ((if (< 4 3) + -) 4 100)

s

cm> (if 0 1 2)

0.2 Warm Up: Write Some Functions

1. Write a function that calculates factorial. (Note we have not seen any iteration yet.)

(define (factorial x) )
2. Write a function that calculates the $n^{th}$ Fibonacci number.

\[
\text{(define (fib n)}
\]
\[
\text{  (if (< n 2) 1
\text{    (define a (cons 3 (cons 2 nil)))
\text{  a
\text{  (car a) 3
\text{  (cdr a) (2
\text{  (car (cdr a)) 2
\text{  (define (len a)}
\text{    (if (null? a) 0
\text{      (+ 1 (len (cdr a)))))
\text{  len
\text{  (len a) 2
\]}

1 Pairs and Lists

To construct a (linked) list in Scheme, you can use the constructor \text{cons} (which takes two arguments). \text{nil} represents the empty list. If you have a linked list in Scheme, you can use selector \text{car} to get the first element and selector \text{cdr} to get the rest of the list. \text{(car and cdr don't stand for anything anymore, but if you want the history go to http://en.wikipedia.org/wiki/CAR_and_CDR.)}

\text{scm> nil}
\text{()}
\text{scm> (null? nil)
#t}
\text{scm> (cons 2 nil)
(2)
scm> (cons 3 (cons 2 nil))
(3 2)
scm> (define a (cons 3 (cons 2 nil)))
\text{a}
\text{scm> (car a) 3
\text{scm> (cdr a) (2
\text{scm> (car (cdr a)) 2
\text{scm> (define (len a)
\text{  (if (null? a) 0
\text{    (+ 1 (len (cdr a)))))
\text{  len
\text{  (len a) 2
}
If a list is a "good looking" list, like the ones above where the second element is always a linked list, we call it a **well-formed list**. Interestingly, in Scheme, the second element does not have to be a linked list. You can give something else instead, but `cons` always takes exactly 2 arguments. These lists are called **malformed list**. The difference is a dot:

```scheme
sce> (cons 2 3)
(2 . 3)
sce> (cons 2 (cons 3 nil))
(2 3)
sce> (cdr (cons 2 3))
3
sce> (cdr (cons 2 (cons 3 nil)))
(3)
```

In general, the rule for displaying a pair is as follows: use the dot to separate the `car` and `cdr` fields of a pair, but if the dot is immediately followed by an open parenthesis, then remove the dot and the parenthesis pair. Thus, `(0 . (1 . 2))` becomes `(0 1 . 2)`

There are many useful operations and shorthands on lists. One of them is **list special form** `list` takes zero or more arguments and returns a list of its arguments. Each argument is in the `car` field of each list element. It behaves the same as quoting a list, which also creates the list.

```scheme
sce> (list 1 2 3)
(1 2 3)
sce> '(1 2 3)
(1 2 3)
sce> (car '(1 2 3))
1
sce> (equal? '(1 2 3) (list 1 2 3))
#t
sce> '(1 . (2 3))
(1 2 3)
sce> '(define (square x) (* x x))
(define (square x) (* x x))
```
1. Define a function that takes 2 lists and concatenates them together. Notice that simply calling `(cons a b)` would not work because it will create a deep list. Instead, think recursively!

```scheme
(define (concat a b)
  ; body of function
)
```

```scheme
scm> (concat '(1 2 3) '(2 3 4))
(1 2 3 2 3 4)
```

2. Define `replicate`, which takes an element `x` and a non-negative integer `n`, and returns a list with `x` repeated `n` times.

```scheme
(define (replicate x n)
  ; body of function
)
```

```scheme
scm> (replicate 5 3)
(5 5 5)
```
3. A run-length encoding is a method of compressing a sequence of letters. The list \((a\ a\ a\ b\ a\ a\ a\ a)\) can be compressed to \(\((a\ 3)\ (b\ 1)\ (a\ 4)\)\), where the compressed version of the sequence keeps track of how many letters appear consecutively.

Write a Scheme function that takes a compressed sequence and expands it into the original sequence. \textit{Hint:} try to use functions you defined earlier in this worksheet.

\begin{verbatim}
(define (uncompress s)
  (let loop ((s s) (res '()))
    (if (null? s) res
        (let ((item (car s)) (rest (cdr s)))
          (match item
            [(list 'a k) (loop rest (cons 'a k res))]
            [(list 'b k) (loop rest (cons 'b k res))]
            [else (loop rest (cons item res))])))

scm> (uncompress '((a 1) (b 2) (c 3)))
(a b b c c c)
\end{verbatim}

### Streams

In Python, we can use iterators to represent infinite sequences. However, Scheme does not support iterators. Let’s see what happens when we try to use a Scheme list to represent an infinite sequence of natural numbers:

\begin{verbatim}
(define (naturals n)
  (cons n (naturals (+ n 1))))
naturals
scm> (naturals 0)
Error: maximum recursion depth exceeded
\end{verbatim}

Because the second argument to \texttt{cons} is always evaluated, we cannot create an infinite sequence of integers using a Scheme list.

Instead, our Scheme interpreter (and scheme.cs61a.org) supports \textit{streams}, which are \textit{lazy} Scheme lists. The first element is represented explicitly, but the rest of the stream’s elements are computed only when needed. This evaluation strategy, where we don’t compute a value until it is needed, is called \textit{lazy evaluation}. Let’s try to implement the sequence of natural numbers again using a stream!
We use the special form \texttt{cons-stream} to create a stream. Note that \texttt{cons-stream} is a special form, because the second operand \texttt{(naturals (+ n 1))} is not evaluated when \texttt{cons-stream} is called. It’s only evaluated when \texttt{cdr-stream} is used to inspect the rest of the stream.

- \texttt{nil} is the empty stream
- \texttt{cons-stream} creates a non-empty stream from an initial element and an expression to compute the rest of the stream
- \texttt{car} returns the first element of the stream
- \texttt{cdr-stream} computes and returns the rest of the stream

Streams are very similar to Scheme lists. The \texttt{cdr} of a Scheme list is either another Scheme list or \texttt{nil}; likewise, the \texttt{cdr-stream} of a stream is either a stream or \texttt{nil}. The difference is that the expression for the rest of the stream is computed the first time that \texttt{cdr-stream} is called, instead of when \texttt{cons-stream} is used. Subsequent calls to \texttt{cdr-stream} return this value without recomputing it. This allows us to efficiently work with infinite streams like the \texttt{naturals} example above. We can see this in action by using a non-pure function to compute the rest of the stream:

\begin{verbatim}
scm> (define (compute-rest n)
    ...> (print 'evaluating!)
    ...> (cons-stream n nil))
compute-rest
scm> (define s (cons-stream 0 (compute-rest 1)))
scm> (car (cdr-stream s))
evaluating!
1
scm> (car (cdr-stream s))
1
\end{verbatim}
Note that the symbol evaluating! is only printed the first time cdr-stream is called, and no other time.

1. Write map-stream, which takes a function \( f \) and a stream \( s \) and returns a new stream, which has all the elements from \( s \), but with \( f \) applied to each one.

\[
(define \ (map-stream \ f \ s))
\]

\[
(scm> \ (define \ evens \ (map-stream \ (lambda \ (x) \ (* \ x \ 2)) \ nat)) \)
\]

\[
(scm> \ (car \ (cdr-stream \ evens)) \)
\]

2. The Fibonacci sequence is a classic infinite sequence. Implement make-fib-stream, which takes two numbers and produces a stream of Fibonacci numbers starting with those two numbers.

\[
(define \ (make-fib-stream \ a \ b))
\]

\[
(scm> \ (define \ fib-stream \ (make-fib-stream \ 0 \ 1)) \)
\]

\[
(scm> \ (car \ (cdr-stream \ (cdr-stream \ (cdr-stream \ (cdr-stream \ (cdr-stream \ fib-stream))))))) \)
\]
Tail-Call Optimization

Scheme implements tail-call optimization, which allows programmers to write recursive functions that use a constant amount of space. A tail call occurs when a function calls another function as its last action of the current frame. Because in this case Scheme won’t make any further variable lookups in the frame, the frame is no longer needed, and we can remove it from memory. In other words, if this is the last thing you are going to do in a function call, we can reuse the current frame instead of making a new frame.

Consider this version of factorial that does not use tail calls:

\[
\begin{align*}
& \text{(define (fact n)} \\
& \quad \text{(if (= n 0) 1} \\
& \quad \quad (* n (fact (- n 1)))))
\end{align*}
\]

The recursive call occurs in the last line, but it is not the last expression evaluated. After calling (fact (- n 1)), the function still needs to multiply that result with n. The final expression that is evaluated is a call to the multiplication function, not fact itself. Therefore, the recursive call is not a tail call.

However, we can rewrite this function using a helper function that remembers the temporary product that we have calculated so far in each recursive step.

\[
\begin{align*}
& \text{(define (fact n)} \\
& \quad \text{(define (fact-tail n result)} \\
& \quad \quad \text{(if (= n 0) result} \\
& \quad \quad \quad \text{(fact-tail (- n 1) (* n result)})}) \\
& \quad \text{(fact-tail n 1))}
\end{align*}
\]

fact-tail makes a single recursive call to fact-tail that is the last expression to be evaluated, so it is a tail call. Therefore, fact-tail is a tail recursive process. Tail recursive processes can take a constant amount of memory because each recursive call frame does not need to be saved. Our original implementation of fact required the program to keep each frame open because the last expression multiplies the recursive result with n. Therefore, at each frame, we need to remember the current value of n.

In contrast, the tail recursive fact-tail does not require the interpreter to remember the values for n or result in each frame. Instead, we can just update the value of n and result of the current frame! Therefore, we can carry out the calculation using only enough memory for a single frame.

### 3.1 Identifying tail calls

A function call is a tail call if it is in a tail context (but a tail call might not be a recursive tail call as seen above in the first fact definition). We consider the following to be tail contexts:
• the last sub-expression in a lambda’s body
• the second or third sub-expression in an if form
• any of the non-predicate sub-expressions in a cond form
• the last sub-expression in an and or an or form
• the last sub-expression in a begin’s body

Before we jump into questions, a quick tip for defining tail recursive functions is to use helper functions. A helper function should have all the arguments from the parent function, plus additional arguments like total or counter or result.

1. For each of the following functions, identify whether it contains a recursive call in a tail context. Also indicate if it uses a constant number of frames.

   (define (question-a x)
      (if (= x 0)
          0
          (+ x (question-a (- x 1)))))

   (define (question-b x y)
      (if (= x 0)
          y
          (question-b (- x 1) (+ y x))))

   (define (question-c x y)
      (if (> x y)
          (question-c (- y 1) x)
          (question-c (+ x 10) y)))

   (define (question-d n)
      (if (question-d n)
          (question-d (- n 1))
          (question-d (+ n 10))))

2. Write a tail recursive function that returns the $n$th fibonacci number. We define $fib(0) = 0$ and $fib(1) = 1$.

   (define (fib n)