SCHEME 10

COMPUTER SCIENCE 61A

July 26, 2016

0.1 Warm Up: Conditional Expressions

1. What does Scheme print?

```
scm> (if (or #t (/ 1 0)) 1 (/ 1 0))

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

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

scm> (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.

```
(define (fib n)
    (if (< n 2)
        1
                                                    )
```

1 Pairs and Lists

To construct a (linked) list in Scheme, you can use the constructor `cons` (which takes two arguments). `nil` represents the empty list. If you have a linked list in Scheme, you can use selector `car` to get the first element and selector `cdr` to get the rest of the list. (`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.)

```
scm> nil
()
scm> (null? nil)
#t
scm> (cons 2 nil)
(2)
scm> (cons 3 (cons 2 nil))
(3 2)
scm> (define a (cons 3 (cons 2 nil)))
a
scm> (car a)
3
scm> (cdr a)
(2)
scm> (car (cdr a))
2
scm> (define (len a)
       (if (null? a)
         0
         (+ 1 (len (cdr a)))))
len
scm> (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:

```
scm> (cons 2 3)
(2 . 3)
scm> (cons 2 (cons 3 nil))
(2 3)
scm> (cdr (cons 2 3))
3
scm> (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.

```
scm> (list 1 2 3)
(1 2 3)
scm> '(1 2 3)
(1 2 3)
scm> (car '(1 2 3))
1
scm> (equal? '(1 2 3) (list 1 2 3))
#t
scm> '(1 . (2 3))
(1 2 3)
scm> '(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!

```
(define (concat a b)
                                                                    )
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.

```
(define (replicate x n)
                                                                  )
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. *Hint:* try to use functions you defined earlier in this worksheet.

```
(define (uncompress s)
                                                        )
scm> (uncompress '((a 1) (b 2) (c 3)))
(a b b c c c)
```

2 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:

```
scm> (define (naturals n)
         (cons n (naturals (+ n 1))))
naturals
scm> (naturals 0)
Error: maximum recursion depth exceeded
```

Because the second argument to `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 *streams*, which are *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 *lazy evalutation*. Let's try to implement the sequence of natural numbers again using a stream!

```
scm> (define (naturals n)
         (cons-stream n (naturals (+ n 1))))
naturals
scm> (define nat (naturals 0))
nat
scm> (car nat)
0
scm> (car (cdr-stream nat))
1
scm> (car (cdr-stream (cdr-stream nat)))
2
```

We use the special form `cons-stream` to create a stream. Note that `cons-stream` is a special form, because the second operand `(naturals (+ n 1))` is *not* evaluated when `cons-stream` is called. It's only evaluated when `cdr-stream` is used to inspect the rest of the stream.

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

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

```
scm> (define (compute-rest n)
...>   (print 'evaluating!)
...>   (cons-stream n nil))
compute-rest
scm> (define s (cons-stream 0 (compute-rest 1)))
s
scm> (car (cdr-stream s))
evaluating!
1
scm> (car (cdr-stream s))
1
```

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))
evens
scm> (car (cdr-stream evens))
2
```

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))
fib-stream
scm> (car (cdr-stream (cdr-stream (cdr-stream (cdr-stream
   (cdr-stream fib-stream))))))
5
```

3 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:

```
(define (fact n)
  (if (= n 0) 1
      (* n (fact (- n 1)))))
```

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.

```
(define (fact n)
  (define (fact-tail n result)
    (if (= n 0) result
        (fact-tail (- n 1) (* n result))))
  (fact-tail n 1))
```

`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 nth fibonacci number. We define fib$(0) = 0$ and fib$(1) = 1$.

```
(define (fib n)
```