This tutorial will cover the following topics:

- Higher order functions including filter, map, foldr, and foldl.
- Using lambda to produce anonymous functions.

For all problems in this tutorial, you may only define helper functions using lambda.
Question 1: Perfect Squares
Using filter and lambda (but no other named helper functions), write a function (perfect-squares lst) which consumes a list of natural numbers (lst) and produces a new list containing only numbers from the original list that are perfect squares, in the same order as they initially appeared.

;; Examples:
(check-expect (perfect-squares (list 2 4 7 8 16)) (list 4 16))
Question 2: Web Search

Web search algorithms use a rich set of features to decide which links are relevant to a given search query (such as “how do I pass CS 135?”). In this problem, you will simulate a web search algorithm. Thankfully, you only must deal with queries that are one word long.

Your search algorithm may encounter two types of documents, either a WebPage or a Video. For this exercise, a WebPage is described by a list of symbols representing the words in the page and a Video by a description (list of symbols) and the number of seconds the video is long. You may use the following data definitions when writing your solution:

;; A Document is a WebPage or a Video.
(define-struct webpage (words))
;; A WebPage is a (make-webpage (ne-listof Sym))

(define-struct video (desc seconds))
;; A Video is a (make-video (ne-listof Sym) Num)
;; Requires: seconds > 0.
Write a function (web-search query docs) which takes a one-word query (represented by a single symbol) and a list of documents (docs) and produces a list of booleans. For each document, the function should produce true if the document is relevant to the query and false otherwise.

- A WebPage is relevant to the query if the exact query symbol is in the list of words on the webpage.
- A Video is relevant to the query if it is at least 5 seconds long and the query symbol is in the list of words in its description.

**Note:** Your solution must use map, cond, and lambda for all helper functions.

;;; Example:

```
(define cat-page (make-webpage (list 'cats 'are 'very 'cool)))
(define dog-page (make-webpage (list 'dogs 'are 'mammals)))
(define cat-video (make-video (list 'video 'of 'my 'cats 'being 'cute) 4))
(check-expect (web-search 'cats (list cat-page dog-page cat-video)) (list true false false))
```
Question 3: Archery Scores
You’ve been asked to automate the scoring process for an upcoming archery competition. In the competition, each archer will shoot multiple rounds of arrows at a target. Each round consists of multiple shots. Each time the archer shoots, their arrow may hit the target or miss. If the arrow hits, it may hit dead center (bullseye) or hit somewhere else on the target (hit).

Write a function (archery-scores rounds) which consumes a list of rounds for one archer in the competition. Each round is a list of symbols, where each symbol represents one of three possible outcomes of one arrow shot by the archer:
  - ‘bullseye, which receives a score of 2
  - ‘hit, which receives a score of 1
  - ‘miss, which receives a score of 0

Rounds do not need to have the same number of shots. The function should produce a single value: the sum of all points received by the given archer.

Note: Your solution must only use cond, lambda, and either foldr or foldl.

;; Example:
(check-expect (archery-score (list (list 'hit 'miss 'bullseye) (list 'hit 'hit)
(list 'miss))) 5)
> The higher order function foldr

(define (my-foldr combine base lst)
  (cond [(empty? lst) base]
          [else (combine (first lst)
                         (my-foldr combine base (rest lst)))]))

foldr is also a built-in function in Intermediate Student With Lambda.
foldr consumes three arguments:

- a function which combines the first list item with the result of reducing the rest of the list;
- a base value;
- a list on which to operate.
(define (my-foldl combine base lst0)
  (local [[(define (foldl/acc lst acc)
              (cond [[(empty? lst) acc]
                     [else (foldl/acc (rest lst)
                                  (combine (first lst) acc))]])]
        (foldl/acc lst0 base))])

(foldl + 0 '(1 2 3 4 5))
The differences between foldr and foldl are:

- the initial value of the accumulator;
- the computation of the new value of the accumulator, given the old value of the accumulator and the first element of the list.

However, there are many cases where they are not interchangeable (e.g., if the combine function is not symmetric, each will produce a different result!).