Problem 1

We can define a position as follows:

```plaintext
data Posn:
    | posn(x :: Number, y :: Number)
end
```

a. Write an expression that constructs a position with an $x$-value of 2 and a $y$-value of 29 and gives it the name $p$.

b. Write an expression that evaluates to the $x$-value of $p$.

c. Write an expression that constructs a new position with each coordinate of $p$ doubled.
Problem 2

Dr Seuss’s book *The Cat in the Hat Comes Back* reveals a new, recursive quality to the Cat in the Hat. When the Cat takes off his hat, Little Cat A is underneath, and under Little Cat A’s hat is Little Cat B, and so on until we find Little Cat Z. Under Little Cat Z’s hat, however, is something else: Voom!\(^1\)

We can represent each of these cats with the following data definition:

```haskell
data Cat:
   | Voom
   | cat(letter :: String, hat :: Cat)
end
```

a. Define three examples of Cats. The first Cat should just be Voom. The second Cat should have Voom in its hat. The third Cat should have the second Cat in its hat.\(^2\)

b. Design the template for a function that takes a Cat as input. You will use this template to design a function next.

---

1 Wondering what Voom is? You’ll need to read the book to find out.
2 That is, the second and third Cats will both be cat structures.
c. Design the function `hat-depth` that takes a `Cat` as input and determines how many hats `Voom` is under. Use the template you just designed to help you get started. Use your three examples of `Cats` to test `hat-depth`. 
Problem 3

Write a function named `sum-large-nums` that takes a list of numbers as input and sums the large numbers in the list, namely those that are $\geq 100$.

Be sure to include a full signature, docstring, and three examples.
Problem 4

Consider this data definition for representing a movie:

```plaintext
data Movie:
    | movie(title :: String, year :: Number, genre :: String, rating :: String)
end
```

a. Define three example movies using this data definition and then create a list containing the movies.

b. Complete this function definition, including examples using your definitions from part (a).

```plaintext
fun count-movie-genre(lom :: List<Movie>, genre :: String) -> Number:
    doc: "Count the number of movies of the given genre in the list"
```
Problem 5

a. First develop the function \texttt{is-odd-single-digit}, which checks whether the number it takes as input is only one digit long and is odd.

\textit{Hint}: To check if a number is odd, you can use \texttt{num-modulo} to find its remainder when divided by 2.

b. Next, use \texttt{is-odd-single-digit} and \texttt{L.filter} (i.e., no recursion) to develop the function \texttt{filter-small-odds}, which takes a list of numbers as input and returns a list of only those elements that were small and odd.
c. Now, consolidate the functions you wrote so that the is-odd-single-digit function is defined inside filter-small-odds.

d. Finally, simplify filter-small-odds by removing the definition of is-odd-single-digit altogether, replacing it with an equivalent lambda (lam) expression.
Problem 6

Recall the abstract, higher-order functions \texttt{L.filter} and \texttt{L.map}. The following list is used below:

\[
\text{NUMS} = \text{[list: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]}
\]

a. Using only the higher-order functions given above, built-in functions, the predicate functions \texttt{is-even} and \texttt{is-odd} (which you can assume have been defined for you), and lambda expression you write yourself, write expressions for the following:

- Double the elements of \texttt{NUMS}.

- Remove all the odd values from \texttt{NUMS}.

- Remove all the even values from \texttt{NUMS}.

b. Evaluate the following expressions:

- \texttt{L.map(num-to-string, NUMS)}

- \texttt{L.filter(lam(x): (x > 5) and (x < 10) end, NUMS)}

- \texttt{L.length(L.filter(lam(x): x == 5 end, NUMS)) > 0}
Problem 7

You are in a maze of twisty little passages, all alike. Consider a world of mazes, which consist of dead ends, represented by a string describing what’s there, or intersections that divide into three mazes, left, middle, and right:

```haskell
data Maze:
    | dead-end(descr :: String)
    | intersection(left :: Maze, middle :: Maze, right :: Maze)
end
```

a. Define four names to represent this maze:

```
START
I1
  "kitty"
  "doldrums"
I3
  "wumpus"
  "candy"
  "bats"
I2
  "treasure"
  "darkness"
  "pit"
  "snakes!"
```

*Hint*: Be careful about the *order* of your statements!

---

3https://en.wikipedia.org/wiki/Colossal_Cave_Adventure
b. Write a function that takes a Maze as input and returns true if it contains the given string:

```python
fun maze-contains(maze :: Maze, goal :: String) -> Boolean:
    doc: "Return true if the maze contains the goal"

where:
    maze-contains(START, "wumpus") is true
    maze-contains(I3, "pit") is false
    maze-contains(START, "waldo") is false
    maze-contains(I2, "snakes!") is true
```