elegant-move

So far we have:

- A state consisting of a snake with a head and a tail, and parameters that control the motion of the head
- A snake head that moves correctly by keyboard inputs
- An initial state that includes a list of positions representing a 3-block long tail. The last item in the tail is the block closest to the head.
- The tails defined by the `List` are displayed by `draw-state` but don't move!

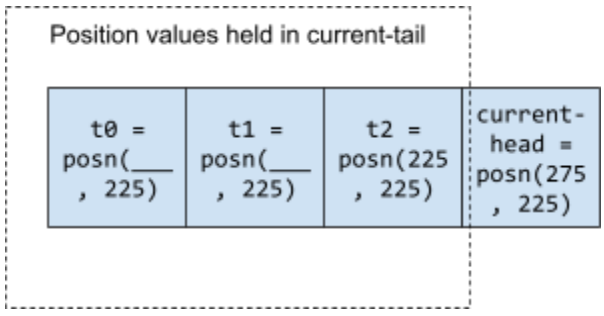
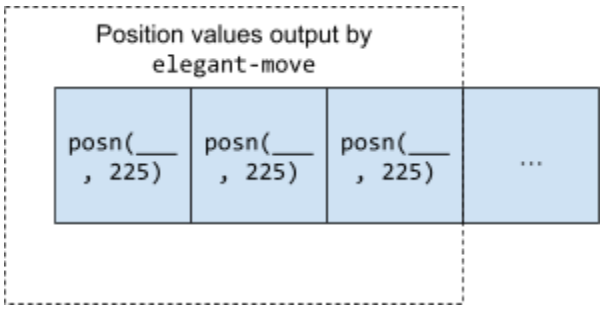
`Problem:`
When the "head" of the snake moves one unit, the list of positions updates so that it moves like a "snake."

The head is moving on its own. It will be helpful to sketch examples with the head in mind, but the function elegant-move will not update the head's motion. We still need the current head position, as we will see.

1. `Sketches`

Look at this drawing of the snake in its current position. The head's next position will be one block to the right.

We are going to write the position list so that the LAST element is nearest the head. (This is the trick that makes it all work elegantly!)

<table>
<tr>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td>Write the list of tail positions before <code>elegant-move</code>:
<code>[list: pos(____ , ____),</code>
<code>pos(____ , ____),</code>
<code>pos(____ , ____)]</code></td>
<td>Write the list of tail positions after:
<code>[list: pos(____ , ____),</code>
<code>pos(____ , ____),</code>
<code>pos(____ , ____)]</code></td>
</tr>
</table>

Write the current-head position: current-head = pos(_____ , _____)	Convince yourself this also works: [list: pos(____ , ____), pos(____ , ____), current-head])

2. Design:

Contract:

elegant-move: current-tail :: _____, current-head:: _____ -> _____

Purpose Statement Restating the Contract:

Examples:

These examples are chosen so that we crunch the whole list, then just the rest. Then repeat for the rest, until we run out of elements.

```
elegant-move(                    is  [list: pos( ____ , ____ ),
[list: pos( ____ , ____ ),                   pos( 225 , 225 ),
          pos( ____ , ____ ),                   pos( ____ , ____)]
             pos(225 , 225)],
       current-head)

elegant-move(                    is  [list:    pos( ____ , ____ ),
[list:    pos( ____ , ____ ),                   pos( ____ , ____)]
             pos(225 , 225)],
       current-head)

elegant-move(                    is  [list: posn(_____ , _____ )]
[list: pos(225 , 225)],
        current-head)

elegant-move(                    is  [list: ]
[list: ], current-head)
```

Go back to the examples and for each input case, circle the `first` element of the input `List` and label it.

Now for each case, circle the `rest` of the input list and label it.

Anywhere you see the *value* of `current-head`, cross it out and write `current-head`.

Are `first` and/or `rest` present in the outputs? Explain:

How can we write each of the answers in terms of `first`, `rest` and `current-head`?
Hint: the method `a.append(b)` *adds* `List b` *to the back of* `List a`.

What if the list is `empty`?

Definition:

```
fun elegant-move(t, h):
      current-tail = t,
      current-head = h,
      cases (t) List:
      |empty => ___________________

      |link(f, r) => ____________________________
      end
end
```

Implement:
To implement this in our code, we'll need to define the function, then figure out how to integrate it into `next-state-tick`. Remember this function takes in an entire game state and returns a new one.