4.3 - A list

Objects, as generic blobs of values, can be used to build all sorts of data structures. A common data structure is the *list* (not to be confused with array). A list is a nested set of objects, with the first object holding a reference to the second, the second to the third, and so on.

```
let list = {
  value: 1,
  rest: {
    value: 2,
    rest: {
      value: 3,
      rest: null
    }
  }
};
```

A nice thing about lists is that they can share parts of their structure. For example, if I create two new values `{value: 0, rest: list}` and `{value: -1, rest: list}` (with `list` referring to the binding defined earlier), they are both independent lists, but they share the structure that makes up their last three elements. The original list is also still a valid three-element list.

Write a function `arrayToList` that builds up a list structure like the one shown when given `[1, 2, 3]` as argument. Also write a `listToArray`function that produces an array from a list. Then add a helper function `prepend`, which takes an element and a list and creates a new list that adds the element to the front of the input list, and `nth`, which takes a list and a number and returns the element at the given position in the list (with zero referring to the first element) or `undefined` when there is no such element. If you haven't already, also write a recursive version of `nth`.

```
// Your code here.
```

```
console.log(arrayToList([10, 20]));
// → {value: 10, rest: {value: 20, rest: null}}
console.log(listToArray(arrayToList([10, 20, 30])));
// → [10, 20, 30]
console.log(prepend(10, prepend(20, null)));
// → {value: 10, rest: {value: 20, rest: null}}
console.log(nth(arrayToList([10, 20, 30]), 1));
// → 20
```

Building up a list is easier when done back to front. So `arrayToList` could iterate over the array backwards (see the previous exercise) and, for each element, add an object to the list. You can use a local binding to hold the part of the list that was built so far and use an assignment like `list = {value: X, rest: list}` to add an element.

To run over a list (in `listToArray` and `nth`), a `for` loop specification like this can be used:

```
for (let node = list; node; node = node.rest) {}
```

Can you see how that works? Every iteration of the loop, `node` points to the current sublist, and the body can read its `value` property to get the current element. At the end of an iteration, `node` moves to the next sublist. When that is null, we have reached the end of the list, and the loop is finished.

The recursive version of `nth` will, similarly, look at an ever smaller part of the "tail" of the list and at the same time count down the index until it reaches zero, at which point it can return the `value` property of the node it is looking at. To get the zeroth element of a list, you simply take the `value` property of its head node. To get element $N + 1$, you take the Nth element of the list that's in this list's `rest` property.