PROBLEM NOTES

Critical to the problem is noticing the following:

- You can't always replicate just the second node passed to fill-nodes. The node to be replicated must have a common parent with the node to its left, in order to include nesting structure, such as the table rows in the third example. If the node passed in doesn't have that property, you have to find an ancestor that does.
- You can't always replicate just the ancestor of the second node either. Any nodes between it and the node to the left needs replicating, e.g., the "," in the second example.

This suggests we need functions that

- take the first and second nodes passed in and return the ancestor nodes (possibly themselves) that have a common parent
- repeatedly copy the nodes from just after the first ancestor through and including the second ancestorB, putting them just after the second ancestor, until there N total copies in the DOM, where N is the number of data items

So, in Lisp, the top-level function might be

```
(defun fill-nodes (nodes texts name)
 (mapcar #'fill-node
   (replicate-nodes nodes (length texts) name)
   texts))
```

The replication function needs to get the sibling parents, replicate as necessary, then return the labeled nodes to fill. It should avoid replication if there's already enough nodes. The question said we could ignore having more nodes than values.

```
(defun replicate-nodes (nodes len name)
  (if (<= len (length nodes))
      nodes
    (multiple-value-bind (a b)
        (get-sibling-ancestors (car nodes) (cadr nodes))
      (dotimes (i (- len (length nodes)))
        (replicate-nodes-between a b))
      (get-nodes (dom-get-parent-node a) name))))
```

Getting the ancestors isn't too tricky, once we get a list of the nodes from the root to each node. `mismatch` is a useful often overlooked function in many libraries:

```
(defun get-sibling-ancestors (a b)
  (let* ((a-ancestors (get-ancestors a))
         (b-ancestors (get-ancestors b))
         (n (mismatch a-ancestors b-ancestors)))
    (values (nth n a-ancestors) (nth n b-ancestors))))
```

```
(defun get-ancestors (node &optional l)
  (cond ((null node) l)
        (t (get-ancestors (dom-get-parent-node node)
                          (cons node l)))))
```

Replicating is tricky. It's easy to get into infinite loops or ill-structured chains. If we have the parents A, B and C, with zero or more "interstitial" nodes between them, e.g., [A, x, y, B, z, w, C], then replicating x, y, B with `insertBefore()` works simplest by going right to left, inserting a copy B' of B before z, a copy y' of y before B' and finally a copy x' of x before y', yielding [A, x, y, B, x', y', B', z, w, C].

The implementation below takes advantage of the fact that `insertBefore()` in the W3C API is defined to return the node inserted.

```
(defun replicate-nodes-between (a b)
  (do ((node b (dom-get-previous-sibling node))
       (ref-node (dom-get-next-sibling b)
                 (dom-insert-before (dom-clone-node node t)
                                    ref-node)))
      ((eql a node) nil)))
```

A complete set of running code in Lisp follows. It includes a quick and dirty implementation of the W3C DOM API, and the three test examples, using the LISP-UNIT package. **All that was being looked for in the qual solution was the above material.**

```
(in-package :cs325-user)


;;; A quick and dirty implementation of DOM nodes for testing purposes.
;;; Pretty standard linked list code.

(defstruct (node
            (:print-function print-node))
  data parent children prev next)

(defun print-node (node out depth)
  (format out "~&~VT#<~S" (* depth 2) (node-data node))
  (dolist (child (node-children node))
    (print-node child out (1+ depth)))
  (format out ">"))

(defun dom-get-parent-node (node) (node-parent node))
(defun dom-get-first-child (node) (car (node-children node)))
(defun dom-get-last-child (node) (car (last (node-children node))))
(defun dom-get-next-sibling (node) (node-next node))
(defun dom-get-previous-sibling (node) (node-prev node))

(defun dom-get-attribute (node attr)
  (and (consp (node-data node))
       (getf (cdr (node-data node)) attr)))

 (defun dom-clone-node (node flag)
  (let ((clone (copy-node node)))
    (cond ((null flag) clone)
          (t (link-children clone
                (mapcar #'(lambda (child) (dom-clone-node child flag))
                        (node-children node)))))))

;;; Set up next/prev and child/parent links, return parent.

(defun link-children (parent children)
  (mapc #'(lambda (node next)
            (setf (node-next node) next
              (node-prev next) node))
    children (cdr children))
  (dolist (child children) (setf (node-parent child) parent))
  (setf (node-children parent) children)
  parent)

 (defun dom-insert-before (node ref-node)
  (setf (node-next node) ref-node
    (node-prev node) (node-prev ref-node)
    (node-parent node) (node-parent ref-node))
  (setf (node-prev ref-node) node)
  (setf (node-children (node-parent node))
    (insert-before node ref-node (node-children (node-parent node))))
  node)

(defun insert-before (a b l)
  (if (eql (car l) b)
```

```
      (cons a l)
    (do ((ll l (cdr ll)))
        ((or (null ll)
             (eql (cadr ll) b))
         (cond ((null ll) (error "missing sibling in parent list"))
               (t (setf (cdr ll) (cons a (cdr ll)))
                  l))))))

;;; Convenience functions for turning an LHTML list into a DOM tree
;;; and back, for testing.

;;; (dom-to-lhtml dom) -> LHTML
;;; (lhtml-to-dom lhtml) -> DOM node

(defun dom-to-lhtml (node)
  (cond ((stringp (node-data node))
         (node-data node))
        (t
         (cons (node-data node)
               (mapcar #'dom-to-lhtml (node-children node))))))

(defun lhtml-to-dom (l)
  (cond ((null l) nil)
        ((atom l) (make-node :data l))
        (t
         (link-children (make-node :data (car l))
                        (mapcar #'lhtml-to-dom (cdr l))))))

;;; The functions provided by the "assistant."

(defun fill-node (node text)
  (setf (node-children node)
        (list (make-node :data text :parent node))))

(defun get-nodes (node name)
  (if (node-p node)
      (let ((nodes (mapcan #'(lambda (child) (get-nodes child name))
                           (node-children node))))
        (if (has-class-p node name)
            (cons node nodes)
          nodes))
    nil))

(defun has-class-p (node name)
  (equal name (dom-get-attribute node :class)))


;;; The qual solution.

(defun fill-nodes (nodes texts)
  (mapcar #'fill-node
    (replicate-nodes nodes (length texts))
    texts))

(defun replicate-nodes (nodes len name)
  (if (<= len (length nodes))
      nodes
```

```
    (multiple-value-bind (a b)
        (get-sibling-ancestors (car nodes) (cadr nodes))
      (dotimes (i (- len (length nodes)))
        (replicate-nodes-between a b))
      (get-nodes (dom-get-parent-node a) name))))

;;; This loop is pretty easy, even if we didn't have Lisp's
;;; mismatch function. More robust code would check for failures
;;; caused by DOM structures not properly arranged for replication.

(defun get-sibling-ancestors (a b)
  (let* ((a-ancestors (get-ancestors a))
         (b-ancestors (get-ancestors b))
         (n (mismatch a-ancestors b-ancestors)))
    (values (nth n a-ancestors) (nth n b-ancestors))))

(defun get-ancestors (node &optional l)
  (cond ((null node) l)
        (t (get-ancestors (dom-get-parent-node node) (cons node l)))))

(defun replicate-nodes-between (a b)
  (do ((node b (dom-get-previous-sibling node))
       (ref-node (dom-get-next-sibling b)
                 (dom-insert-before (dom-clone-node node t)
                                    ref-node)))
      ((eql a node) nil)))


;;; Test the 3 examples in LHTML form

;;; A convenience function for test calls.

(defun fill-lhtml (lhtml name texts)
  (let ((node (lhtml-to-dom lhtml)))
    (fill-nodes (get-nodes node name) texts)
    (dom-to-lhtml node)))

(define-test simple-list
  (assert-equal '(p ((span :class "score") "10")
                    ((span :class "score") "15")
                    ((span :class "score") "20")
                    ((span :class "score") "25")
                    ((span :class "score") "30"))
                (fill-lhtml '(p ((span :class "score") 1)
                                ((span :class "score") 2)
                                ((span :class "score") 3))
                            "score"
                            '("10" "15" "20" "25" "30"))))


(define-test and-list
  (assert-equal '(p "Scores:"
                    ((span :class "score") "10") ", "
                    ((span :class "score") "15") ", "
                    ((span :class "score") "20") ", "
                    ((span :class "score") "25") " and "
                    ((span :class "score") "30") ". ")
```

```
                (fill-lhtml '(p "Scores:"
                                ((span :class "score") 1) ", "
                                ((span :class "score") 2) " and "
                                ((span :class "score") 3) ". ")
                            "score"
                            '("10" "15" "20" "25" "30"))))

(define-test table-list
  (assert-equal '(table
                  (tr (th "Scores"))
                  (tr ((td :class "score") "10"))
                  (tr ((td :class "score") "15"))
                  (tr ((td :class "score") "20"))
                  (tr ((td :class "score") "25"))
                  (tr ((td :class "score") "30"))
                  )
                (fill-lhtml '(table
                              (tr (th "Scores"))
                              (tr ((td :class "score") "1"))
                              (tr ((td :class "score") "2"))
                              (tr ((td :class "score") "3"))
                              )
                            "score"
                            '("10" "15" "20" "25" "30"))))

;;; An extra example, not in qual, with uneven nesting. In
;;; this case, B is more deeply nested than A.

(define-test uneven-list
  (assert-equal '(p "Scores:"
                    ((span :class "score") "10") ", "
                    ((font :color "red")
                     (b ((span :class "score") "15"))) ", "
                    ((font :color "red")
                     (b ((span :class "score") "20"))) ", "
                    ((font :color "red")
                     (b ((span :class "score") "25"))) " and "
                    ((span :class "score") "30") ". ")
                (fill-lhtml '(p "Scores:"
                                ((span :class "score") 1) ", "
                                ((font :color "red")
                                 (b ((span :class "score") 2))) " and "
                                ((span :class "score") 3) ". ")
                            "score"
                            '("10" "15" "20" "25" "30"))))
```