

MASSACHUSETTS INSTITUTE OF TECHNOLOGY  
 Department of Electrical Engineering and Computer Science  
 6.001—Structure and Interpretation of Computer Programs  
 Spring 2004

**Recitation 18**  
**Quiz 2 Review**

## Problems

### 1. Mutation

```
(define x 1)
(set! x (cons x x))
(set! x (cons x x))
(set-cdr! (car x) x)
(set-car! (cddr x) (cadr x))
x
```

Draw box-and-pointer diagram for `x`.

2. **Trie implementation** - Used for string searching. Looks like a binary tree, but each node has up to  $\Sigma$  children, where  $\Sigma$  is size of the alphabet. Each child pointer is labelled with the character.

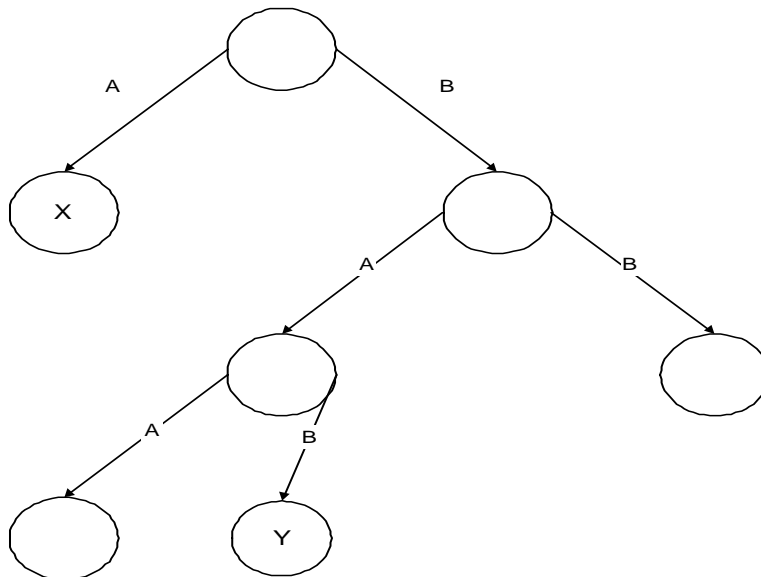


Figure 1: Example trie: value of key (a) is X; value of key (b a b) is Y.

In our implementation, we'll represent a string key as a list of single-character symbols: "hello" = '(h e l l o). In order to look up a key in the trie, start at the root node and follow the appropriately labelled child pointers until you reach the end of the key. To insert a

new `<key,value>` pair, follow key until you reach the end of the trie, then create child nodes until the key is empty, finally store the value at the last node created.

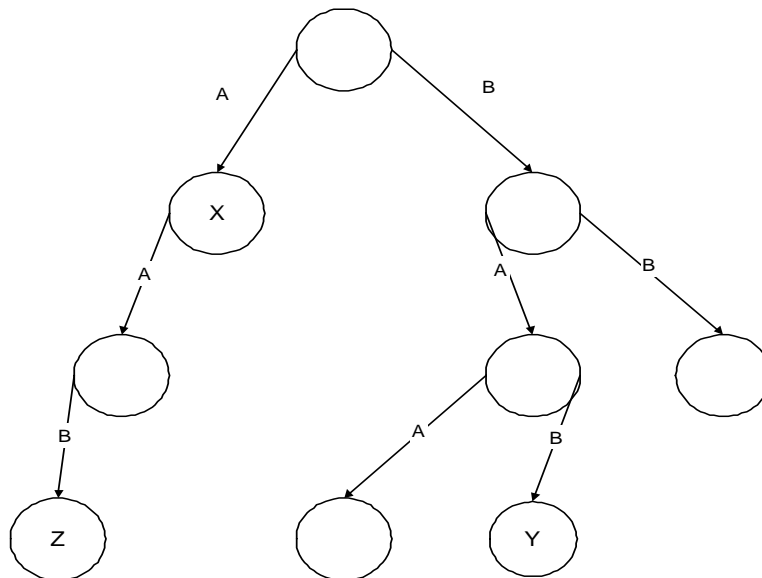


Figure 2: Example trie: insert `<(aab,Z)>` into previous trie.

- (a) Implement `make-node` which builds a trie node. A node has a value and an initially empty set of children. This should be implemented as a tagged data structure.

```
(define (make-node node)
```

- (b) Implement `trie-node?` which returns `#t` if it is passed a trie node as input.

```
(define (trie-node? x)
```

- (c) Implement `node-value` which takes a node and returns the node's value.

```
(define (node-value node)
```

- (d) Implement `node-child` which takes an item (a one character symbol) and a node, and returns the child of the node labelled with item.

```
(define (node-child item node)
```

```
(define (trie-lookup key node)
  (if (null? key)
      (node-value node)
      (let ((child (node-child (car key) node)))
        (if child
            (trie-lookup (cdr key) child)
            #f))))
```

- (e) Implement `trie-insert!`, which takes a key (list of items), a value, and the root node of the trie to insert into. Subsequent `trie-lookups` on key should yield the value. Any intermediate nodes created should have the default value `#f`.

```
(define (trie-insert! key value node)
```

### 3. Environment-model

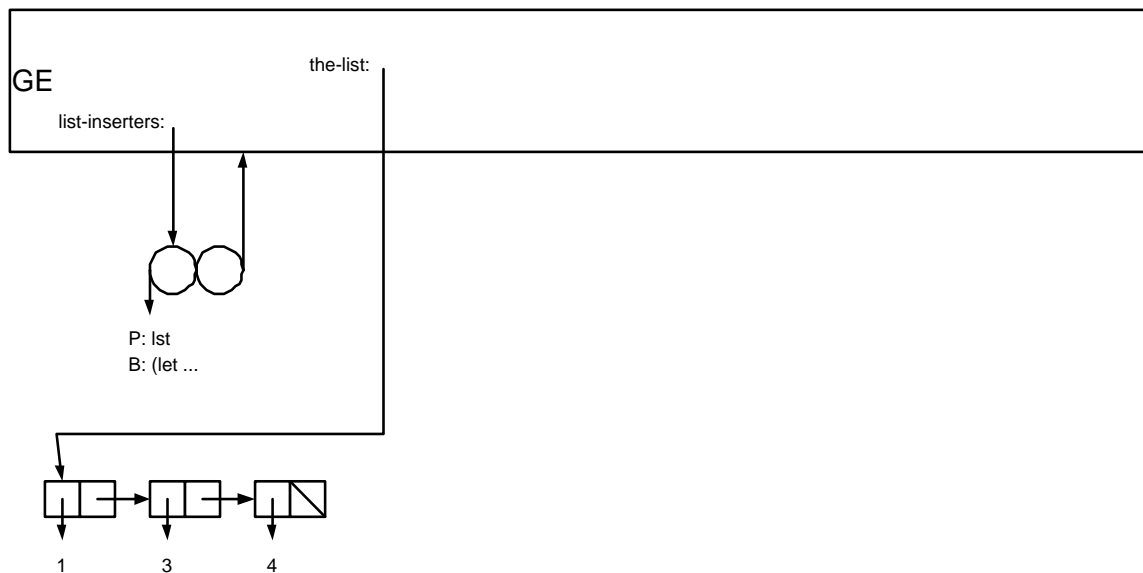
The procedure `last-pair` returns the last pair of a list (guaranteed to have `nil` in the `cdr`).

```
(define (list-inserters lst)
  (let ((last (last-pair lst)))
    (list (lambda (x)
            (set-cdr! lst (cons x (cdr lst)))
            lst)
          (lambda (y)
            (set-cdr! last (cons y nil))
            (set! last (cdr last))
            lst))))))
```

```
(define the-list (list 1 3 4))
```

```
(let ((ins (list-inserters the-list)))
  ((first ins) 2)
  ((second ins) 5))
```

Finish the environment diagram.



#### 4. Object-Oriented Programming

Re-implement the trie node data structure from problem 2 as an object.

- (a) Implement the `create-node` procedure.

```
(define (create-node value)
```

- (b) Write out the skeleton of the `make-node` procedure (no methods other than the `TYPE` method).

- (c) Implement the `VALUE` method.

- (d) Implement the `CHILD` method, which takes in a label and returns the child with that label or `#f`.

- (e) Implement the `SET-VALUE!` method, which is a mutator for the value.

- (f) Implement the `ADD-CHILD!` method, which takes in a label and a newnode, and adds the newnode as a child of current node with the given label.

- (g) Implement the `LOOKUP` method, which takes in a key and acts like the previous `trie-lookup` procedure.

- (h) Implement the `INSERT!` method, which works like the previous `trie-insert!` except it calls `SET-VALUE!` and `ADD-CHILD!` where appropriate.

```
(define (make-node self value)
```

They're coming to take you away.. Ha Ha!

## Feedback

Year:                      Programming Experience:                      Favorite Color:  
Section: 4 or 6

1. In general, how is recitation:  
Great!      Good      OK      Poor      I don't attend
2. Recitation pace compared to your optimal pace?  
Too Fast      Fast      OK      Slow      Zzzzzz
3. Problem difficulty?  
Too hard      OK      Too easy
4. Any comments / suggestions for improvement:

5. Did you have fun with project 4?