MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring 2006

Recitation 19 — 4/21/2006
Quiz II Review

1. **Symbols & Quote**

   Write the results of evaluating these expressions:

```
(list '(a) '(b))

(append '(c) '((d)))

(quote (list))
```

2. **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`.

3. **Environment Diagrams**

   Evaluate the following expressions and write their resulting values. (Draw an environment diagram as you go.)

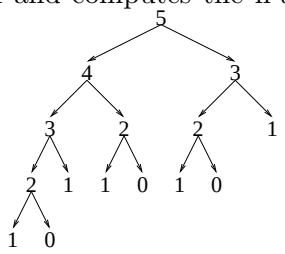
```
(define (f)
   (let ((x 0))
     (lambda (y)
       (set! x (+ x 1))
       (* x y))))

((f) 1)
((f) 2)
```

   How many double-bubbles are created in the evaluation of these expressions (including the temporary double-bubbles used by let)?

4. **Counting Fibs**

Recall the function fib-1 that takes an integer n and computes the n'th Fibonacci number.

```
(define fib-1
  (lambda (n)
    (cond ((= n 0) 0)
          ((= n 1) 1)
          (else
           (+ (fib-1 (- n 1))
              (fib-1 (- n 2)))))))
```



What is the Order of Growth in time for the procedure fib-1? This is a tough one to figure out. Maybe this tree will help. Consider the number of recursive calls to fib-1 when the following is evaluated: (fib-1 5)

What if we want to see exactly how many times fib-1 is being called? Recall the function make-count-proc from last section. How can we use make-count-proc to define fib-2 that will keep a count of the number of recursive calls?

```
(define make-count-proc
  (lambda (f)
    (let ((count 0))
      (lambda (x)
        (cond ((eq? x 'count) count)
              ((eq? x 'reset)
               (set! count 0)
               0)
              (else
               (set! count (+ count 1))
               (f x)))))))
```

```
(define fib-2
```

```
(fib-2 5) ==> 5
(fib-2 'count ==> ?
```

Take a look at these calls to fib-2:

```
(fib-2 'reset)        ==> 0
(fib-2 30)            ==> 832040
(fib-2 'count)        ==> 2692537
```

That's pretty inefficient! We're recursively calling fib-2 over and over again with the same argument and keep computing things we've already computed before. How can we fix this?

5. **Memoizing**

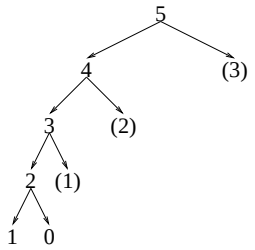
Recall that the procedure make-count-proc takes in a procedure and returns a very similar procedure (from the caller's point of view), but this new procedure keeps some local state around and does something else each time it is called.

Consider the procedure memoize that takes in a procedure of one argument and returns a procedure that keeps track of the all previously computed values. If a value passed in was passed in before, the procedure simply returns the saved value. Write the procedure memoize:

```
(define memoize
  (lambda (g)
    (let ((table '()))
      (lambda (y)
        (let ((result (assoc y table)))
          (if (pair? result)
              (cadr result)
              (let ((result (g y)))
                (set! table (cons (list y result) table))
                result)))))))
```

Now define fib-3 that uses memoization and a counter.

```
(define fib-3
```



What is the Order of Growth in Time of fib-3?

```
(fib-3 'count)                ==> 0
(fib-3 30)                    ==> 832040
(fib-3 'count)                ==> 59
```

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

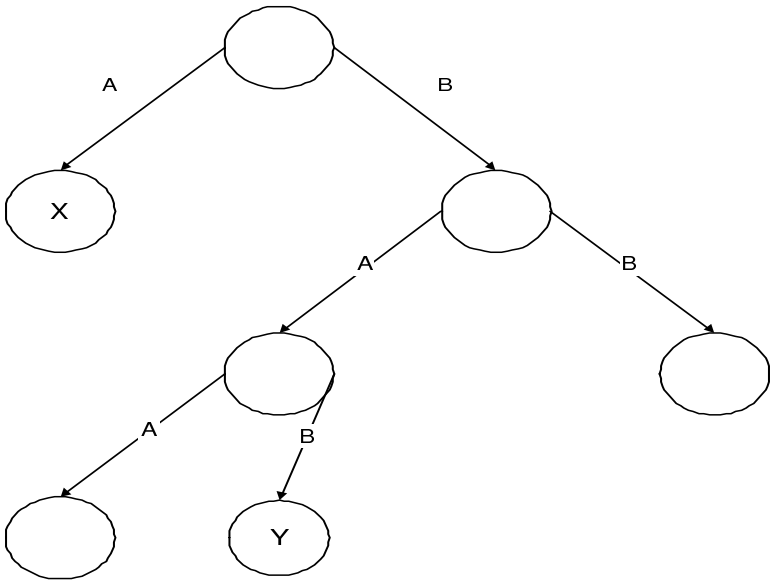


Figure 1: Example trie: value of key `(a)` is X; value of key `(b a b)` is Y.

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.

(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)
```

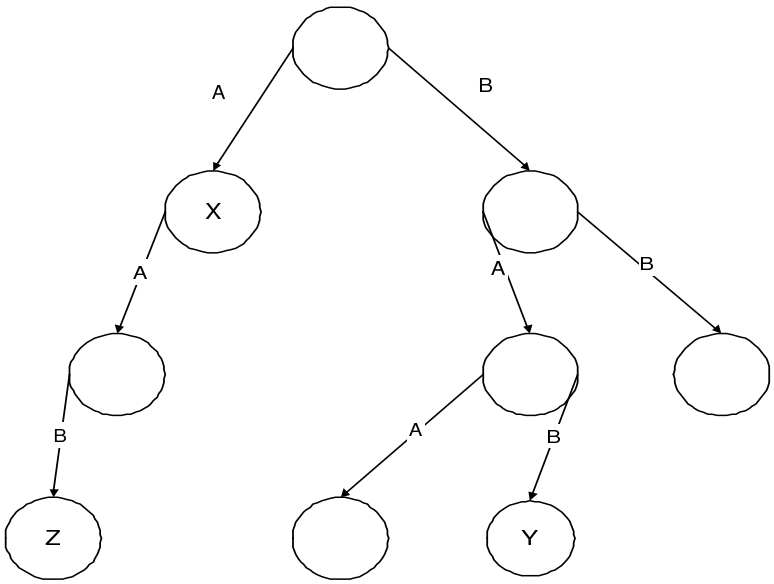


Figure 2: Example trie: insert `<(aab,Z)>` into previous trie.

(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-lookup`s on key should yield the value. Any intermediate nodes created should have the default value `#f`.

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