

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

**Recitation 12**  
**ADTs: Tables**

## Scheme

### 1. Procedures

- (a) (`assq key alist`) - Searches through *alist* looking for element whose car is *key*. If found, it returns the whole element, otherwise `#f`. Comparisons are done with `eq?`.
- (b) (`assv key alist`) - Same as `assq` except it uses `eqv?` for key comparison.
- (c) (`assoc key alist`) - Same as `assq` except it uses `equal?` for key comparison.

## Problems

### Truth tables

Input 1	Input 2	Output
#t	#t	#t
#t	#f	#f
#f	#t	#f
#f	#f	#f

Input 1	Input 2	Output
#t	#t	#t
#t	#f	#t
#f	#t	#t
#f	#f	#f

1. Write a procedure `lookup` that, given two inputs and a truth table, looks up the output.

```
(define (lookup t1 t2 lookup-table)
```

**Gates and Circuit simulation**

```
(load-option 'hash-table)

; globals table
(define globals
  (make-eq-hash-table))

(define (get-counter)
  (hash-table/get globals 'counter 0))

(define (inc-counter!)
  (hash-table/put! globals 'counter
    (+ 1 (get-counter))))

; component-table abstraction
(define component-table
  (make-eqv-hash-table))

(define (component-table-put! key elem)
  (hash-table/put! component-table key elem))

(define (component-table-get key)
  (hash-table/get component-table key #f))

(define (component-table-keys)
  (hash-table/key-list component-table))

(define (component-table-clear!)
  (hash-table/clear! component-table))
```

Components have: input1,input2,output,output-side, lookup-table.

2. Write `make-component`, which takes a `lookup-table` of the function it implements and returns the number assigned to the component.

```
(define (make-component lookup-table)
```

3. Write `component-get`, which returns a property of a component given its number. If the component doesn't have that property, return the symbol `empty`.

```
(define (component-get num property)
```

With these, the selectors and mutators of the component are easy:

```
(define (component-input1 num)
  (component-get num 'input1))
```

```
(define (component-input2 num)
  (component-get num 'input2))
```

```
(define (component-output-num num)
  (component-get num 'output))
```

```
(define (component-output-side num)
  (component-get num 'output-side))
```

```
(define (component-lookup-table num)
  (component-get num 'lookup-table))
```

```
(define (component-print num)
  (pp (hash-table->alist (component-table-get num))))
```

5. Write `component-connect!`, which given two component numbers and a side, connects the output of the first component to the side of the second component.

```
(define (component-connect! cnum1 cnum2 side)
```

6. Complete `component-output-data`.

```
(define (component-output-data num data)
  (let ((output (component-output-num num))
        (output-side (component-output-side num)))
    (cond ((or (eq? output 'empty) (eq? output-side 'empty))
           (error "unconnected component output" num))
          ((eq? output 'output)
           (display "Result: ")
           (display data)
           (newline)
           #t)
          (else
           #f))))
```

7. Write `component-process`, which given a component number computes the output if the inputs are available. It should return true if it successfully displayed a result.

```
(define (component-process num)
```

8. Write `simulate`, which attempts to process each component, stopping when at least one component successfully displays a result.

```
(define (simulate)
```