Scheme

1. Procedures

   (a) \( \texttt{(assq key alist)} \) - Searches through \( \texttt{alist} \) looking for element whose car is \( \texttt{key} \). If found, it returns the whole element, otherwise \#f. Comparisons are done with \texttt{eq?}.

   (b) \( \texttt{(assv key alist)} \) - Same as \( \texttt{assq} \) except it uses \texttt{eqv?} for key comparison.

   (c) \( \texttt{(assoc key alist)} \) - Same as \( \texttt{assq} \) except it uses \texttt{equal?} for key comparison.

2. Table Abstraction

   (a) \( \texttt{(make-table)} \) - creates a table.

   (b) \( \texttt{(table-get table key default-value)} \) - If \( <\texttt{key,value}> \) is in the table, returns \texttt{value}. Otherwise returns \texttt{default-value}.

   (c) \( \texttt{(table-put! table key value)} \) - Inserts \( <\texttt{key,value}> \) into the table, updating \texttt{value} if the \texttt{key} is already in the table.

   (d) \( \texttt{(table-data table)} \) - Returns a list of the values in the table.

   (e) \( \texttt{(table-clear! table)} \) - Removes all data from the table.

Problems

Truth tables

<table>
<thead>
<tr>
<th>Input 1</th>
<th>Input 2</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>#t</td>
<td>#t</td>
<td>#t</td>
</tr>
<tr>
<td>#t</td>
<td>#f</td>
<td>#f</td>
</tr>
<tr>
<td>#f</td>
<td>#t</td>
<td>#f</td>
</tr>
<tr>
<td>#f</td>
<td>#f</td>
<td>#f</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input 1</th>
<th>Input 2</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>#t</td>
<td>#t</td>
<td>#t</td>
</tr>
<tr>
<td>#t</td>
<td>#f</td>
<td>#t</td>
</tr>
<tr>
<td>#f</td>
<td>#t</td>
<td>#t</td>
</tr>
<tr>
<td>#f</td>
<td>#f</td>
<td>#f</td>
</tr>
</tbody>
</table>
1. Write a procedure `lookup` that, given a list of inputs and a truth table, looks up the output.

   (define (lookup inputs lookup-table)

Gates and Circuit simulation

   (define (make-component name table inputs)
     (list 'component name table inputs))
   (define (component-name component)
     (second component))
   (define (component-table component)
     (third component))
   (define (component-inputs component)
     (fourth component))
   (define (component? x)
     (and (pair? x) (eq? (car x) 'component)))

   Suppose we store the components in a table:

   (define table-of-components (make-table))

2. Implement the procedure `add-component!`, which makes a component from its input and adds it to the table using the component name as the key.

   (define (add-component! name table inputs)

3. Implement the procedure `all-components`, which returns a list of all the components in the table.

   (define (all-components)
In order to simulate the network, let’s use a “blackboard” to store values produced by components. When a component produces a value, it writes the value on the blackboard under its name. In order for a component to produce an output, the blackboard must have values for each of its inputs. The blackboard will be implemented as a table:

```
(define output-table (make-table))

(define (set-output! name value)
  (table-put! output-table name value))

(define (get-output name)
  (table-get output-table name 'not-ready)) ; returns not-ready if value is not on the blackboard

(define (clear-output!)
  (table-clear! output-table))
```

4. When a component successfully processes its inputs and produces an output, it needs to write this output to the blackboard. Write a procedure `component-output-data`, which takes a component and the data it wants to output, and puts this data on the blackboard under the component’s name.

```
(define (component-output-data component data)
```

5. Finish the procedure `component-process`, which tests to see if the inputs for the component are available, and if so computes and writes out the output of the component.

```
(define (component-process component)
  (let ((inputs  
  (if (memq 'not-ready inputs) ; returns #t if not-ready in list of inputs 'nothing-to-process
```

Here’s a piece of code that processes every component until an output is written on the blackboard under a particular name:

```
(define (step output)
  (if (eq? (get-output output) 'not-ready)
    (begin
      (map component-process (all-components))
      (step output))
    (get-output output)))
```
6. Write `simulate`, which should do the following things: clear the blackboard, write the inputs on the blackboard, and then start stepping through the code. Remember that procedure bodies may contain multiple statements. Here's an example usage:

```
(add-component! 'D not-table '(A))
(add-component! 'E and-table '(B D))
(add-component! 'F or-table '(C E))

(simulate '((A #f) (B #t) (C #f)) 'F)
```

```
(define (simulate inputs output)

MITScheme built-in hash-table implementation of tables:

(define make-table make-eq-hash-table)
(define table-get hash-table/get)
(define table-put! hash-table/put!)
(define table-data hash-table/datum-list)
(define table-clear! hash-table/clear!)

Alternative association list implementation:

(define (make-table)
  (list 'atable))
(define (table-put! table key value)
  (set-cdr! table (cons (list key value) (cdr table))))
(define (table-clear! table)
  (set-cdr! table nil))

7. Write `table-get` for the association list implementation:

```
(define (table-get table key default-value)
```

8. Write `table-data`:

```
(define (table-data table)
```