Scheme

1. Procedures

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

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

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

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 \text{lookup} that, given two inputs and a truth table, looks up the output.

\[
\text{(define (lookup t1 t2 lookup-table)}
\]

(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.

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

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

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

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

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

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

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

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

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

   ```scheme
   (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.

   ```scheme
   (define (component-connect! cnum1 cnum2 side)
   ```
6. Complete component-output-data.

```scheme
(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.

```scheme
(define (component-process num)
  (component-output-data num data))
```

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

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