Modeling with Mutable Data

SICP section 3.3

3.3.1 Mutable List Structure

Introducing primitive mutators: set-car! and set-cdr!

cons creates a new list:

- `list a: (1 2 3)`
- `list b: (4 5 6)`
- `(cons a b)`
- `list c: (1 2 3 4 5 6)`

set-car! and set-cdr! change the original list:

- `(set-car! a (car b))`
- `list a: (4 2 3)`
- `list b: (4 5 6)`

3.3.2 Representing Queues

What is a queue?

- FIFO buffer – insert data at the rear, delete from the front

Basic operations:

- `(make queue)` – constructor
- `(empty-queue? ⟨queue⟩)` and `(front-queue ⟨queue⟩)` – selectors
- `(insert-queue! ⟨queue⟩ ⟨item⟩)` and `(delete-queue! ⟨queue⟩)` – mutators

Use set-car! and set-cdr! to set the front and rear pointers in a queue:

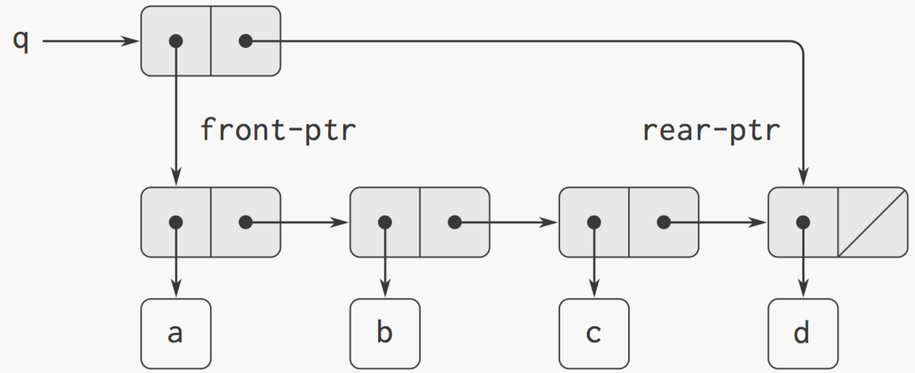
- `(define (set-front-ptr! queue item) (set-car! queue item))`
- `(define (set-rear-ptr! queue item) (set-cdr! queue item))`

```
(define (insert-queue! queue item)
  (let ((new-pair (cons item '())))
    (cond ((empty-queue? queue)
           (set-front-ptr! queue new-pair)
           (set-rear-ptr! queue new-pair)
           queue)
          (else (set-cdr! (rear-ptr queue)
                          new-pair)
                (set-rear-ptr! queue new-pair)
                queue))))
```

```
(define (make-queue) (cons '() '()))
```

```
(define (empty-queue? queue)
  (null? (front-ptr queue)))
```

```
(define (front-queue queue)
  (if (empty-queue? queue)
      (error "FRONT called with
        an empty queue" queue)
      (car (front-ptr queue))))
```




```
(define (delete-queue! queue)
  (cond ((empty-queue? queue)
         (error "DELETE! called with
                 an empty queue" queue))
        (else (set-front-ptr! queue
                (cdr (front-ptr queue)))
              queue)))
```

3.3.3 Representing Tables

How to represent a one-dimensional table:

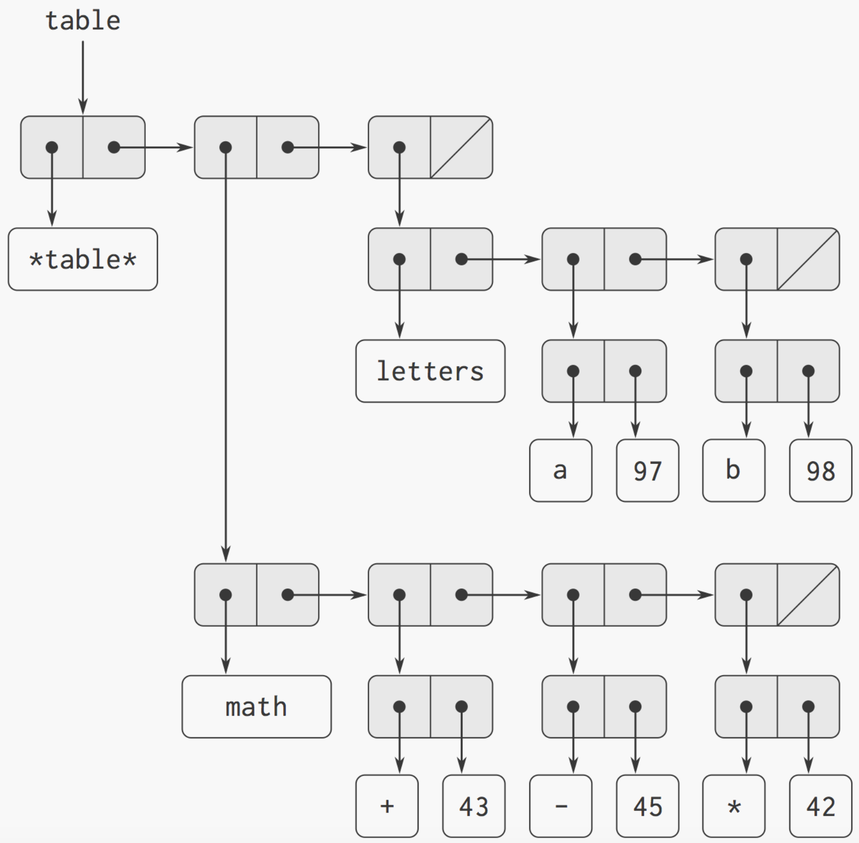
- Headed list of key-value pairs (backbone)
- Has a special “dummy” backbone pair at the beginning
- In each pair, the car points to the successive record

How to represent a two-dimensional table:

- Same as one-dimensional table, but each value has two keys (key-subtable pairs)

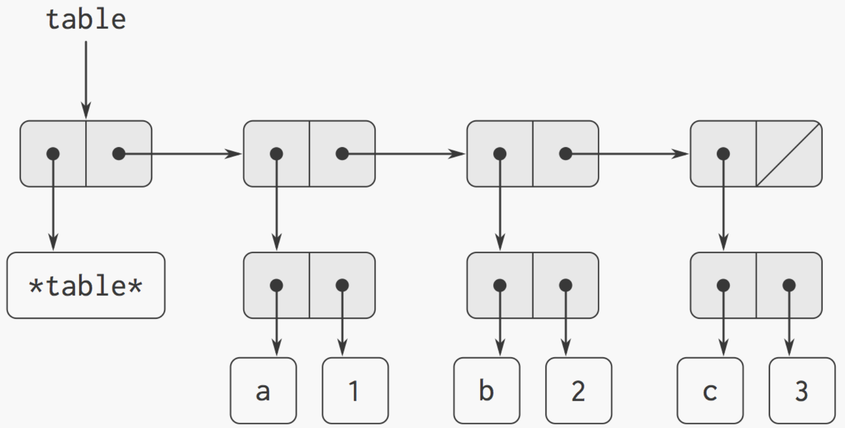
```
math:   +: 43     letters:   a: 97
        -: 45                b: 98
        *: 42
```

<– Two-dimensional table




```
a:   1
b:   2
c:   3
```

<– One-dimensional table



3.3.4 A Simulator for Digital Circuits

A digital circuit is made of:

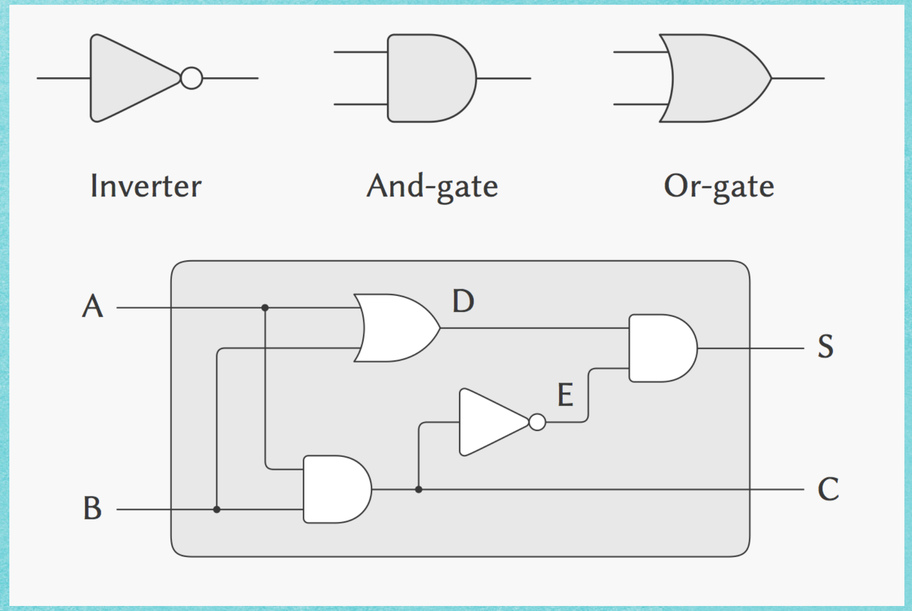
- Wires, carrying digital signals, either 0 or 1
- Function boxes, connecting wires and computing the output

Operations on primitive function boxes:

- `(get-signal ⟨wire⟩)`
- `(set-signal! ⟨wire⟩ ⟨new value⟩)`
- `(add-action! ⟨wire⟩ ⟨procedure of no arguments⟩)`

Operations on wires (same as queues):

- `(make-agenda)`
- `(empty-agenda? ⟨agenda⟩)`
- `(first-agenda-item ⟨agenda⟩)`
- `(remove-first-agenda-item! ⟨agenda⟩)`
- `(add-to-agenda! ⟨time⟩ ⟨action⟩ ⟨agenda⟩)`
- `(current-time ⟨agenda⟩)`



A little digression: Q-Lisp

Quantum computing:

- Qubits (atoms, ions, photons…) instead of bits
- Can be either 0, 1, or both (superposition)
- Make measurements using entanglement
- Inherent parallelism – millions of times more powerful than a supercomputer

Q-Lisp Project:

- Extension of Common Lisp
- Processing data on machines with non-deterministic and quantum registers
- See the Q-Lisp Project and the Black Stone Project (on GitHub)

3.3.5 Propagation of Constraints

Introducing constraint programming:

- Allows us to perform multi-directional computations (e.g. equations)
- Constraints – primitive elements defining relations between quantities
- Constraint networks – made of constraints joined by connectors

Constraint system implementation (same as circuits):

- `(has-value? ⟨connector⟩)`
- `(get-value ⟨connector⟩)`
- `(set-value! ⟨connector⟩ ⟨new-value⟩ ⟨informant⟩)`
- `(forget-value! ⟨connector⟩ ⟨retractor⟩)`
- `(connect ⟨connector⟩ ⟨new-constraint⟩)`

Conclusion

The primary benefit of computer languages may not be that we can program computers with them, but that they force us to systematize our understanding of the world to the point where a digital computer can emulate it - Gerald J. Sussman

Links:

See this lecture by Sussman

Thanks!