Exercise 2.29.

A binary mobile consists of two branches, a left branch and a right branch. Each branch is a rod of a certain length, from which hangs either a weight or another binary mobile. We can represent a binary mobile using compound data by constructing it from two branches (for example, using `list`):

```
(define (make-mobile left right)
  (list left right))
```

A branch is constructed from a `length` (which must be a number) together with a `structure`, which may be either a number (representing a simple weight) or another mobile:

```
(define (make-branch length structure)
  (list length structure))
```

a. Write the corresponding selectors `left-branch` and `right-branch`, which return the branches of a mobile, and `branch-length` and `branch-structure`, which return the components of a branch.

b. Using your selectors, define a procedure `total-weight` that returns the total weight of a mobile.

c. A mobile is said to be `balanced` if the torque applied by its top-left branch is equal to that applied by its top-right branch (that is, if the length of the left rod multiplied by the weight hanging from that rod is equal to the corresponding product for the right side) and if each of the submobiles hanging off its branches is balanced. Design a predicate that tests whether a binary mobile is balanced.

d. Suppose we change the representation of mobiles so that the constructors are

```
(define (make-mobile left right)
  (cons left right))

(define (make-branch length structure)
  (cons length structure))
```

How much do you need to change your programs to convert to the new representation?

Answer.

a. A typical mobile `m` can be expressed in box-and-pointer notation like figure 1. Thus, we can imme-

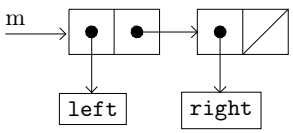

Figure 1. The box-and-pointer notation of a mobile

diately write the selectors `left-branch` and `right-branch`:

```
(define (left-branch m)
  (car m))

(define (right-branch m)
  (car (cdr m)))
```

Similarly, the corresponding selectors `branch-length` and `branch-structure` can be expressed as:

```
(define (branch-length b)
  (car b))

(define (branch-structure b)
  (car (cdr b)))
```

b. As an example shown in figure 2, the `total-weight` of a mobile can be computed in a recursive plan. The reduction step is:

- The `total-weight` of a mobile `m` is the `total-weight` of the `left-branch` of `m` plus the `total-weight` of the `right-branch` of `m`.

This is applied successfully until we reach the base case:

- The `total-weight` of a number is the value of that number.

Finally, to make the program robust, we'd better also take the empty list into consideration:

- The `total-weight` of an empty list is 0.

Hence, these reveal the whole picture of `total-weight`[1]:

```
(define (total-weight m)
  (cond ((null? m) 0)
        ((not (pair? m)) m)
        (else
         (+ (total-weight (branch-structure (left-branch m)))
            (total-weight (branch-structure (right-branch m)))))))
```

c. Given the description of a balanced mobile, the predicate to test whether a moblile is balanced can be implemented as:

```
(define (balanced? m)
  (and (= (torque (left-branch m))
          (torque (right-branch m)))
       (branch-balanced? (left-branch m))
       (branch-balanced? (right-branch m))))
```

where the `torque` is a procedure to compute the product of the length and the weight of a branch

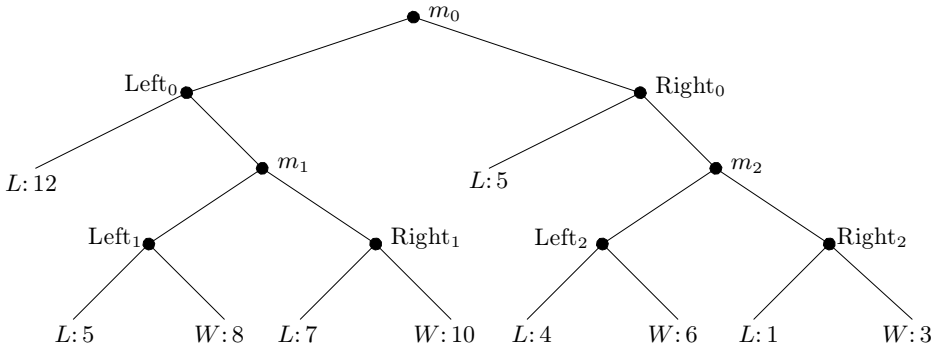

Figure 2. A example of mobile

1. Besides, there is another way to get the `total-weight` of a mobile. Here we introduce a procedure to obtain the weight of a branch, namely `branch-weight` and make `total-weight` and `branch-weight` invoke each other crossedly

```
(define (total-weight m)
  (cond ((null? m) 0)
        ((not (pair? m)) m)
        (else
          (+ (branch-weight (left-branch m))
             (branch-weight (right-branch m))))))

(define (branch-weight b)
  (if (not (pair? (branch-structure b)))
      (branch-structure b)
      (total-weight (branch-structure b))))
```

```
(define (torque b)
  (* (branch-length b)
     (branch-weight b)))

(define (branch-weight b)
  (if (not (pair? (branch-structure b)))
      (branch-structure b)
      (+ (branch-weight (left-branch (branch-structure b)))
         (branch-weight (right-branch (branch-structure b))))))
```

To test a branch is balanced, do the following:

- If the `structure` of a branch `b` is a number rather than a mobile, then, the branch is balanced.
- Otherwise, test whether the mobile of that branch is balanced or not.

```
(define (branch-balanced? b)
  (if (not (pair? (branch-structure b)))
      #t
      (balanced? (branch-structure b))))
```

d. Note that in designing this system to deal with the object mobile, we mainly used data abstraction to help us control its complexity. The abstraction barriers we established to isolate the operation on mobile with its underlying representation are the constructor and selectors: `make-mobile`, `left-branch` and `right-branch`. Similar is the data object branch whose constructor and selectors are `make-branch`, `branch-length` and `branch-structure`. Only these constructors and selectors know the underlying implementation of a mobile and a branch. We can envision the structure of mobile system as shown in figure 3. In this case, only the procedure `right-branch` and `branch-structure` are affected when

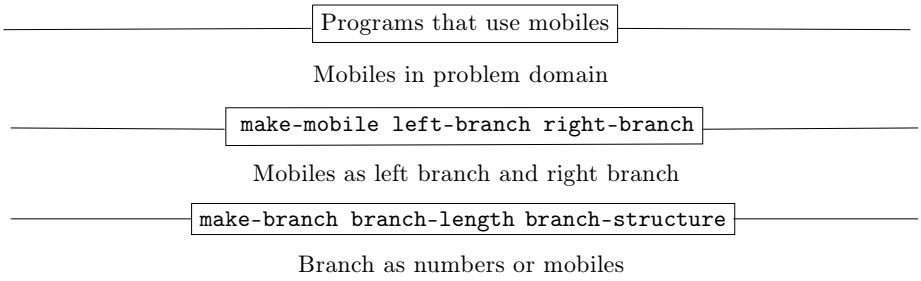

Figure 3. Data-abstraction barriers in the mobile package

the representation of mobile changes. Hence, the program needs only a minor adjustment:

```
(define (right-branch m)
  (cdr m))

(define (branch-structure b)
  (cdr b))
```