Chapter 2: Building Abstractions with Data

Using simple data types in complex procedures bounds our possibilities - that's why we now want to abtract the data too! We call it compound data and increase our modularity by a lot.

A simple example is addition of rationals, where we got a numerator and a denominator. Keeping track which is which in complex procedures can be tricky and thus a compound data object, which has both in it, makes it much easier to handle.

It's important to "generalize" the primitive operators related to abstract data like here:

```
(define (linear-combination a b x y)
(+ (* a x) (* b y)))

->

(define (linear-combination a b x y)
(add (mul a x) (mul b y)))
```

Also, we need to define *symbolic expressions*, which means data, which isn't bound by numbers but also allows characters. We need *generic operations* (like *add* and *mul* above), which act as a kind of template for the parameters.

2.1 Introduction to Data Abstraction

With data abstraction, the compound data object is isolated from the details of how it is constructed from primitive data objects! We then got an abstract data type which is integrated by a set of procedures: selectors and constructors, which implement the abstract data in a concrete representation.

2.1.1 Arithmetic Operations for Rational Numbers

Assuming we already got following procedures

> `(make-rat <n> <d>)` returns the rational number whose numerator is the integer `<n>` and whose denominator is the integer `<d>`.
>
> `(numer <x>)` returns the numerator of the rational number `<x>`.
>
> `(denom <x>)` returns the denominator of the rational number `<x>`.

With these procedures we need to define all the math operations we desire:

```
(define (add-rat x y)
(make-rat (+ (* (numer x) (denom y))
(* (numer y) (denom x)))
(* (denom x) (denom y))))
(define (sub-rat x y)
(make-rat (- (* (numer x) (denom y))
(* (numer y) (denom x)))
(* (denom x) (denom y))))
(define (mul-rat x y)
(make-rat (* (numer x) (numer y))
(* (denom x) (denom y))))
(define (div-rat x y)
(make-rat (* (numer x) (denom y))
(* (denom x) (numer y))))
(define (equal-rat? x y)
(= (* (numer x) (denom y))
(* (numer y) (denom x))))
```

These are basic fractional math rules. Now we need the definitions of the constructor of the rationals:

In Lisp one can create a *pair* with cons, and access first and second with *car* and *cdr*.

```
> (define x (cons 1 2))
> (car x)
> 1
> (cdr x)
> 2
```

Of course we can create pairs out of pairs and so on, to further generalize the data. These are called *list-structured data*.

Defining a rational is easy now:

```
(define (make-rat x y) (cons x y))
(define (numer r) (car r))
(define (denom r) (cdr r))
```

One could make the constructor smarter by always finding the gcd and then creating the rational.

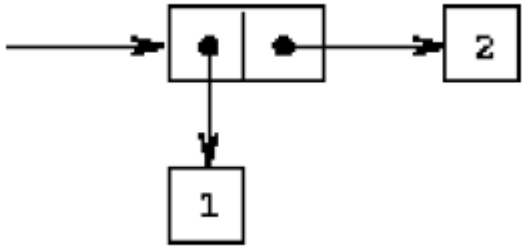
```
(define (make-rat n d)
(let ((g (gcd n d)))
(cons (/ n g) (/ d g))))
```

2.1.2 Abstraction Barriers

Every abstraction layer has its operators defined in their layer, they are seperated by abstraction barriers. This makes a program way more testable and easier to modify. The details how pairs are implemented are not interesting in the layer of rationals etc.

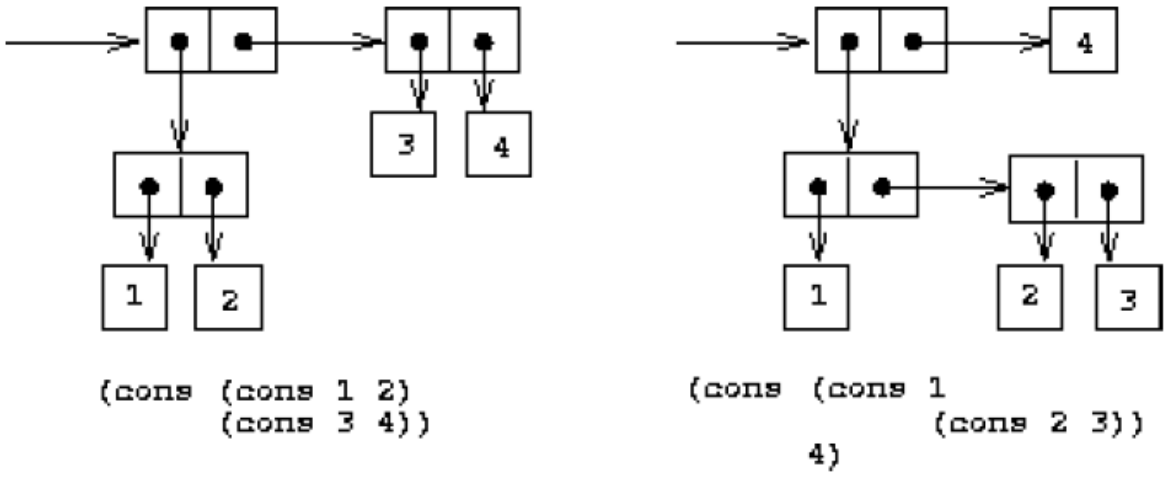
2.2 Hierarchical Data and the Closure Property

In a box-and-pointer notation, the objects are shown as pointers to a box. The box contains a representation of the object.



An example for a box-and-pointer for a pair

If one makes pairs out of pairs, it will look f.e. like



Different ways of defining a pair of pairs

cons has a *closure property* which means that we can combine things and these operations can be combined the same using the same operation. This allows to create hierarchical strucutres.

2.2.1 Representing Sequences

It's straight forward to define sequences with pairs:

```
(cons 1
  (cons 2
    (cons 3
      (cons 4 nil))))
```

The pointer will always point to the next pair, until nil (nihil → nothing) is reached. These nested constructions are called a *list*. Of course they are also defineable in Lisp as

```
(list <a1> <a2> ... <an>)
```

List operations

If one would like to get the n'th element of a list, one could define f.e. the procedure

```
(define (list-ref items n)
  (if (= n 0) (car items)
  (list-ref (cdr items) (- n 1))))
```

Mapping over lists

We can f.e. transform each element of the list with simple procedures:

```
(define (scale-list item factor)
  (if (null? item) nil
    (cons (* (car item) factor)
      (scale-list (cdr item) factor)
```

Going for higher-level abstraction we get

```
(define (map item term)
  (if (null? item) nil
    (cons (term (car item))(map (cdr item) term) ) ) )
```

where we use a procedure as an input. We can now call the procedure f.e. with

```
(map (list 1 2 3 4) lambda(x) (* x x) ) )

> (1 4 9 16)
```

2.2.2 Hierarchical Structures