Till now we have operated on simple numerical data. However, simple data types are not sufficient for many of the problems we wish to address using computation.

Another key aspect of any programming language is the means it provides for building abstractions by combining data objects to form compound data.

A natural language would consist of several simple words. However, if there are no means to combine these words to form a sentence, to utility of any natural language would be fairly limited.

Why do we want compound data in a programming language?

Why do we want compound data in a programming language?

For the same reasons that we want compound procedures:

- **to elevate the conceptual level at which we can design programs.**
  Just as
  the ability to define procedures enables us to deal with processes at a higher conceptual level than that of the primitive operations of the language,
  the ability to construct compound data objects enables us to deal with data at a higher conceptual level than that of the primitive data objects of the language.

  $\rightarrow$ A car ('data') can be considered as a collection of metal pieces (strings/integers), but it would be easier to consider a car as a single entity at a higher conceptual level as we always expect all parts of the car to be moving together.

- **to increase the modularity of our designs,** and
  We would ideally like to task of building a car to be seperate from the task of running a car. The car driver need not know about the internal mechanisms of the car to be able to drive the car.

- **to enhance the expressive power of our language.**
  As long as the driver (say subroutine) is able to drive the car (data), the passenger (main procedure) in the back seat, should not be to expected to know how to drive a car.

Data abstraction is a methodology that enables us to isolate how a compound data object is used from the details of how it is constructed from more primitive data objects.

Data abstraction is a methodology that enables us to isolate how a compound data object is used from the details of how it is constructed from more primitive data objects.

The basic idea of data abstraction is to structure the programs that are to use compound data objects so that they operate on "abstract data."

Data abstraction makes programs much easier to design, maintain, and modify.

We also need a "concrete" data representation that is defined independent of the programs that use the data.

We also need a "concrete" data representation that is defined independent of the programs that use the data.

The interface between these two parts of our system will be a set of procedures, called **selectors and constructors**, that implement the abstract data in terms of the concrete representation.

Suppose we want to do arithmetic (say add, multiply, check equality) with rational numbers.

Assumptions

- We have a way of constructing a rational number from a numerator and a denominator.
- Given a rational number, we have a way of extracting (or selecting) its numerator and its denominator.
- The constructor and selectors are available as 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.

In this case, make-rat is the constructor, while the other two are selectors.

- (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.

In this case, make-rat is the constructor, while the other two are selectors.

Wishful thinking allows us to assume that such representations exist.

Assuming the existence of these procedures we can build procedures for performing arithmetic operations on rational numbers.

```
(define (add-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 (equal-rat? x y)
  (= (* (numer x) (denom y))
     (* (numer y) (denom x))))
```

Now we need to find some way to glue together a numerator and a denominator to form a rational number.
There may be several ways to represent a rational number.
We have **isolated** how a rational number is used from the details of how it is constructed.
Scheme provides a compound structure to glue two entities using **pairs** which can be constructed with the primitive procedure cons.

The cons procedure takes two arguments and returns a compound data object that contains the two arguments as parts. Given a pair, we can extract the parts using the primitive procedures car and cdr.

```
> (cons  1 2)
(1 . 2)
> (car (cons 1 2))
1
> (cdr (cons 1 2))
2
```

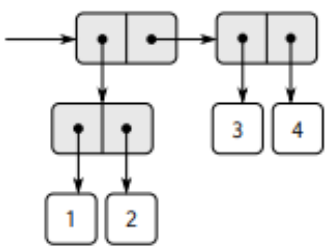
A list is a collection of pairs, such that the last pair has a null/ () as its second element.

```
> (cons 1 null)
(1)
> (cons 1 '())
(1)
> (define t (cons 1 null))
> (car t)
1
> (cdr t)
()
```

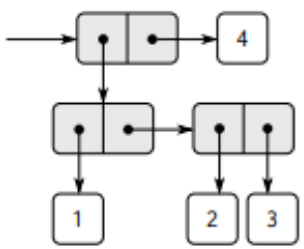
if y is not a pair,
(cons x y) returns (x . y)

```
> (cons (cons 1 2) 3)
((1 . 2) . 3)
> (cons (cons 1 2) (cons 3 null))
((1 . 2) 3)
> (cons (cons 1 2) (cons 3 4))
((1 . 2) 3 . 4)
> (cons (cons 1 (cons 2 3)) 4)
((1 2 . 3) . 4)
```

else it returns a pair which has x inserted into y as its first element.
The dot would be missing between x and y.



(cons (cons 1 2)
(cons 3 4))



(cons (cons 1
(cons 2 3))
4)

In general, the rule for printing a pair is as follows: use the dot notation always, but if the dot is immediately followed by an open parenthesis, then remove the dot, the open parenthesis, and the matching close parenthesis.

Thus,

(0 . (1 . 2)) becomes (0 1 . 2), and

(1 . (2 . (3 . ()))) becomes (1 . (2 . (3))).

which becomes (1 . (2 3)).

which becomes (1 2 3).

The ability to combine pairs means that pairs can be used as general-purpose building blocks to create all sorts of complex data structures.

The single compound-data primitive *pair*, implemented by the procedures cons, car, and cdr, is the only glue we need.

The ability to combine pairs means that pairs can be used as general-purpose building blocks to create all sorts of complex data structures.

The single compound-data primitive *pair*, implemented by the procedures cons, car, and cdr, is the only glue we need.

Data objects constructed from pairs are called *list-structured* data.

```
(define (make-rat n d) (cons n d))
(define (numer x) (car x))
(define (denom x) (cdr x))
```

Now, we can use pairs to construct rational numbers and select the numerator/denominator of a rational number.

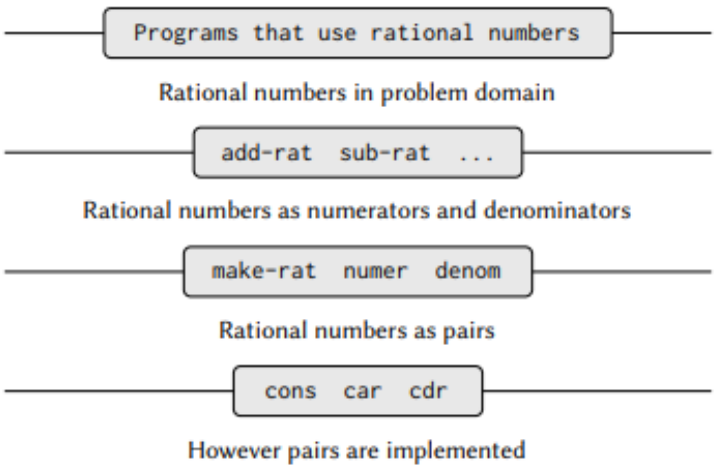
```
(define (print-rat x)
  (newline)
  (display (numer x))
  (display "/")
  (display (denom x)))
```

```
> (define x (make-rat 5 6))
> (define y (make-rat 2 3))
> (print-rat x)
5/6
> (mul-rat x y)
(10 . 18)
> (print-rat (mul-rat x y))
10/18
```

The multiplication of rational numbers has not been presented as an irreducible fraction, however one can insert another procedure to ensure that make-rat always produces irreducible fractions.

You can now use this idea to represent combinations of simple objects like point in 2-dimensions and use combination of two points in 2-dimensions to represent a line-segment or complex numbers with a real and imaginary part.

Or you could also imagine this pairs to be a row in a data base and then create multiple rows to create a database.



In general, the underlying idea of data abstraction is to identify for each type of data object a basic set of operations in terms of which all manipulations of data objects of that type will be expressed, and then to use only those operations in manipulating the data.

In effect, procedures at each level are the interfaces that define the abstraction barriers and connect the different levels.

The horizontal lines represent abstraction barriers that isolate different "levels" of the system. At each level, the barrier separates the programs (above) that use the data abstraction from the programs (below) that implement the data abstraction

Data abstraction makes programs much easier to maintain and to modify.

Without the data abstraction, the procedures would depend on how the data is represented in the memory and one would not have the freedom to change the representation without changing constructors and selectors used by the procedure.

For example, One could implement a gcd procedure within the make-rat (constructors) or one could implement a gcd procedure within the numer and denom procedures (selectors).
This choice of representation does not affect the rest of the procedures which manipulate rational numbers like add-rat, sub-rat.

Right now, we are implementing the notion of rational numbers using pairs and extracting its various parts using selectors.
We could think of these operations as being defined in terms of data objects— numerators, denominators, and rational numbers—whose behavior was specified by the latter three procedures.

Right now, we are implementing the notion of rational numbers using pairs and extracting its various parts using selectors.
We could think of these operations as being defined in terms of data objects— numerators, denominators, and rational numbers—whose behavior was specified by the latter three procedures.

But what is data?
Is data something which is implemented by the given selectors and constructors?
As long as x is (make-rat n d) and $\frac{(\text{numer } x)}{(\text{denom } x)} = \frac{n}{d}$

In general, we can think of data as defined by

- some collection of selectors and constructors,
- together with specified conditions that these procedures must fulfill in order to be a valid representation.

This point of view can serve to define not only "high-level" data objects, such as rational numbers, but lower-level objects as well.

Consider the notion of a pair, which we used in order to define our rational numbers.

In general, we can think of data as defined by

- some collection of selectors and constructors,
- together with specified conditions that these procedures must fulfill in order to be a valid representation.

This point of view can serve to define not only "high-level" data objects, such as rational numbers, but lower-level objects as well.

Consider the notion of a pair, which we used in order to define our rational numbers.

We never actually said what a pair was, only that the language supplied procedures cons, car, and cdr for operating on pairs. But the only thing we need to know about these three operations is that if we glue two objects together using cons we can retrieve the objects using car and cdr.

```
(define (cons a b)
        (lambda (choice)
           (cond((= choice 1) a)
                ((= choice 2) b)))
)


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

This use of procedures corresponds to nothing like our intuitive notion of what data should be.

```
(define (cons a b)
        (lambda (choice)
           (cond((= choice 1) a)
                ((= choice 2) b)))
)


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

This use of procedures corresponds to nothing like our intuitive notion of what data should be.

Nevertheless, all we need to do to show that this is a valid way to represent pairs is to verify that these procedures satisfy the condition given above.

These cons, car and cdr procedures are not the primitives procedures although they behave exactly like primitive procedures as far as their usage for the rational numbers is concerned.

```
13 (define x (cons 5 4))
14 (display "x is ") (display x)
15 (newline)
16 (display "car of x is ") (display (car x))
17 (newline)
18 (display "cdr of x is ") (display (cdr x))
19
```

```
Welcome to DrRacket, version 8.7 [cs].
Language: scheme, with debugging; memory limit: 128 MB.
x is #<procedure:...ta-or-procedure.rkt:4:8>
car of x is 5
cdr of x is 4
>
```

The subtle point to notice is that the value returned by (cons a b) is a procedure.

This procedural implementation of pairs is a valid implementation, and if we access pairs using only cons, car, and cdr we cannot distinguish this implementation from one that uses "real" data structures.

This is not how Scheme works, however this is one way scheme could have worked.

This procedural implementation of pairs is a valid implementation, and if we access pairs using only cons, car, and cdr we cannot distinguish this implementation from one that uses "real" data structures.

This is not how Scheme works, however this is one way scheme could have worked.

This example also demonstrates that the ability to manipulate procedures as objects automatically provides the ability to represent compound data.

This style of programming, which uses procedural representations of data is often called **message passing**.