

Data abstraction

- Reminder: primitive expressions, means of combination, means of abstraction
- The same for data: primitive data, compound data, data abstraction

Let's pretend!

- Pretend that Scheme only has integers and real numbers, no rationals or complex numbers
- We will define our own implementation of rational numbers and complex numbers
- Illustrates data abstraction, multiple representation

Data abstraction

- Methodology that combines many data objects so that they can be treated as one data object
- The new data objects are abstract data: they are used without making any assumptions about how they are implemented
- Data abstraction: define **representation**, hide with **selectors** and **constructors**
- Extends the programming language

Language extensions for handling abstract data

- **Constructor**: a procedure that creates instances of abstract data from data that is passed to it
- **Selector**: a procedure that returns a component datum that is in an abstract data object
- The component datum returned might be the value of an internal variable, or it might be computed

Rational numbers

- Mathematically represented by a pair of integers: $1/2$, $56/874$, $78/23$, etc.
- Constructor:
`(make-rat numerator denominator)`
- Selectors: `(num r)`, `(denom r)`
- That's all a user needs to know!

User defines operations on rational numbers!

Add:

$$n_1/d_1 + n_2/d_2 = (n_1*d_2 + n_2*d_1)/(d_1*d_2)$$

Rational addition

```
(define (add-rat x y)
  (make-rat (+ (* (num x)
                  (denom y))
              (* (num y)
                  (denom x)))
            (* (denom x)
              (denom y))))
```

Another operation

Multiply:

$$(n_1/d_1) * (n_2/d_2) = (n_1*n_2) / (d_1*d_2)$$

Rational multiplication

```
(define (mul-rat x y)
  (make-rat (* (number x)
               (number y))
            (* (denom x)
               (denom y))))
```

A test

Equality:

$$n_1/d_1 = n_2/d_2 \text{ iff } n_1 * d_2 = n_2 * d_1$$

Rational equality

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

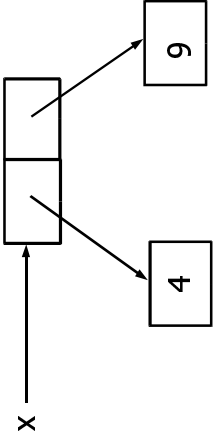
Subtraction and division defined similarly to addition and multiplication

Compound data structures in Scheme

- Called a **pair**
 - Constructor is **cons**
 - Selectors are **car** and **cdr**
- ```
(define x (cons 4 9))
(car x) --> 4
(cdr x) --> 9
```

## Pairs as records with two fields

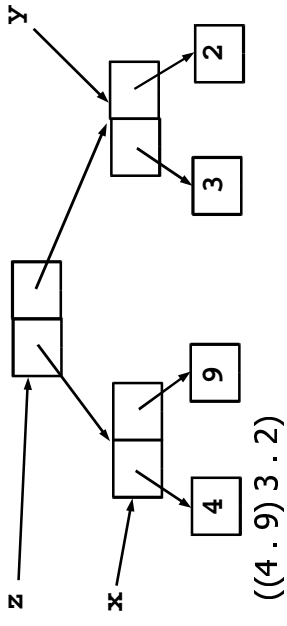
```
(define x (cons 4 9)) produces
```



(4 . 9)

## Building a larger data structure

```
(define y (cons 3 2))
(define z (cons x y))
```



((4 . 9) 3 . 2)

## Extracting data

```
(car (car z)) --> 4
(car (cdr z)) --> 3
(cdr (car z)) --> 9
(cdr (cdr z)) --> 2
```

## List structures

Any data structure built using cons

Lists are a subset of the possible list structures

None of the list structures on the last three slides are lists

## Representing rational numbers

```
(define (make-rat n d) (cons n d))
(define (numer x) (car x))
(define (denom x) (cdr x))
(define (print-rat x)
 (display (numer x))
 (display "/")
 (display (denom x))
 (newline))
```

## Using rational numbers

```
(define one-third (make-rat 1 3))
(define four-fifths
 (make-rat 4 5))
(print-rat one-third)
1/3
(print-rat (add-rat one-third
 four-fifths))
17/15
```

## Some more rational numbers

```
(print-rat (mul-rat one-third
 four-fifths))
4/15
(print-rat (add-rat four-fifths
 four-fifths))
40/25
```

## To get the standard representation

```
(define (make-rat n d)
 (let ((g (gcd n d)))
 (cons (/ n g) (/ d g))))
(print-rat (add-rat four-fifths
 four-fifths))
8/5
```

## Levels of abstraction

- Programs are built up as layers of language extensions
- Each layer is a level of abstraction
- Each level hides some implementation details
- There are four levels of abstraction in our rational numbers example

## Bottom level

- level of pairs
- procedures `cons`, `car` and `cdr` are already provided in the programming language
- The actual implementation of pairs is hidden

## Second level

- Level of rational numbers as data objects
- Procedures `make-rat`, `numer` and `denom` are defined at this level
- Actual implementation of rational numbers is hidden at this level

## Third level

- Level of service procedures on rational numbers
- Procedures `add-rat`, `mul-rat`, `equal-rat`, etc. are defined at this level
- Implementation of these procedures are hidden at this level

## Top level

- Program level
- Rational numbers are used in calculations as if they were ordinary numbers

## Abstraction barriers

- Each level is designed to hide implementation details from higher-level procedures
- These levels act as **abstraction barriers**

## Advantages of data abstraction

- Programs can be designed one level of abstraction at a time
- We don't have to be aware of implementation details below the level at which we are programming
- This means there is less to keep in mind at any one time while programming
- An implementation can be changed later without changing procedures written at higher levels

## Example of changing an implementation

```
(define (make-rat n d) (cons n d))
(define (number x)
 (let ((g (gcd (car x) (cdr x))))
 (/ (car x) g)))
(define (denom x)
 (let ((g (gcd (car x) (cdr x))))
 (/ (cdr x) g)))
```

## Another advantage

- Data abstraction supports top-down design
- We can gradually figure out representations, constructors, selectors and service procedures that we need, one level at a time