Scheme Data Abstraction

Class outline:

- Data abstraction
- Rational abstraction
- Tree abstraction

Data abstraction

Data abstractions

Many values in programs are compound values, a value composed of other values.

- A date: a year, a month, and a day
- A geographic position: latitude and longitude

Scheme does not support OOP or have a dictionary data type, so how can we represent compound values?

A **data abstraction** lets us manipulate compound values as units, without needing to worry about the way the values are stored.

A pair abstraction

If we needed to frequently manipulate "pairs" of values in our program, we could use a `pair` data abstraction.

`(pair a b)`	constructs a new pair from the two arguments.
`(first pair)`	returns the first value in the given pair.
`(second pair)`	returns the second value in the given pair.

```
(define couple (pair 'neil 'david))

(first couple)    ; 'neil
(second couple)   ; 'david
```

A pair implementation

Only the developers of the `pair` abstraction needs to know/decide how to implement it.

```
(define (pair a b)

)

(define (first pair)

)

(define (second pair)

)
```

How else could it be implemented?

A pair implementation

Only the developers of the `pair` abstraction needs to know/decide how to implement it.

```
(define (pair a b)
     (cons a (cons b '()))
)

(define (first pair)

)

(define (second pair)

)
```

How else could it be implemented?

A pair implementation

Only the developers of the `pair` abstraction needs to know/decide how to implement it.

```
(define (pair a b)
    (cons a (cons b '()))
)

(define (first pair)
    (car pair)
)

(define (second pair)

)
```

How else could it be implemented?

A pair implementation

Only the developers of the `pair` abstraction needs to know/decide how to implement it.

```
(define (pair a b)
     (cons a (cons b '()))
)

(define (first pair)
     (car pair)
)

(define (second pair)
     (car (cdr pair))
)
```

How else could it be implemented?

Rational abstraction

Rational numbers

If we needed to represent fractions exactly...

$$\frac{numerator}{denominator}$$

We could use this data abstraction:

Constructor	`(rational n d)`	constructs a new rational number.
Selectors	`(numer r)`	returns the numerator of the given rational number.
	`(denom r)`	returns the denominator of the given rational number.

```
(define quarter (rational 1 4))
(numer quarter) ; 1
(denom quarter) ; 4
```

Rational number arithmetic

Example	General form
$\frac{3}{2} \times \frac{3}{5} = \frac{9}{10}$	$\frac{n_x}{d_x} \times \frac{n_y}{d_y} = \frac{n_x \times n_y}{d_x \times d_y}$
$\frac{3}{2} + \frac{3}{5} = \frac{21}{10}$	$\frac{n_x}{d_x} + \frac{n_y}{d_y} = \frac{n_x \times d_y + n_y \times d_x}{d_x \times d_y}$

Rational number arithmetic code

We can implement arithmetic using the data abstractions:

Implementation	General form
`(define (mul-rational x y) (rational (* (numer x) (numer y)) (* (denom x) (denom y))))`	$\frac{n_x}{d_x} \times \frac{n_y}{d_y} = \frac{n_x \times n_y}{d_x \times d_y}$

```
(define (mul-rational x y)
    (rational
        (* (numer x) (numer y))
        (* (denom x) (denom y))
    )
)
```

$$\frac{n_x}{d_x} \times \frac{n_y}{d_y} = \frac{n_x \times n_y}{d_x \times d_y}$$

```
(mul-rational (rational 3 2) (rational 3 5))   ; (9 10)
```

Rational number arithmetic code

We can implement arithmetic using the data abstractions:

Implementation	**General form**

```
(define (add-rational x y)
    (define nx (numer x))
    (define dx (denom x))
    (define ny (numer y))
    (define dy (denom y))
    (rational
        (+ (* nx dy) (* ny dx) )
        (* dx dy)
    )
)
```

$$\frac{n_x}{d_x} + \frac{n_y}{d_y} = \frac{n_x \times d_y + n_y \times d_x}{d_x \times d_y}$$

```
(add-rational (rational 3 2) (rational 3 5))  ; (21 10)
```

Rational numbers utilities

```
(define (print-rational x)
    (print (numer x) '/ (denom x))
)
```

```
(print-rational (rational 3 2) )  ; 3 / 2
```

Rational numbers utilities

```
(define (print-rational x)
     (print (numer x) '/ (denom x))
)
```

```
(print-rational (rational 3 2) )   ; 3 / 2
```

```
(define (rationals-are-equal x y)
     (and
          (= (* (numer x) (denom y))
             (* (numer y) (denom x))
          )
     )
)
```

```
(rationals-are-equal (rational 3 2) (rational 6 4) ) #t
(rationals-are-equal (rational 3 2) (rational 3 2) ) #t
(rationals-are-equal (rational 3 2) (rational 1 2) ) #f
```

Rational numbers implementation

```
; Construct a rational number that represents N/D
(define (rational n d)
    (list n d)
)

; Return the numerator of rational number R.
(define (numer r)
    (car r)
)

; Return the denominator of rational number R.
(define (denom r)
    (car (cdr r))
)
```

Reducing to lowest terms

What's the current problem with...

```
(add-rational (rational 3 4) (rational 2 16) )   ; 56/64
(add-rational (rational 3 4) (rational 4 16) )   ; 64/64
```

Reducing to lowest terms

What's the current problem with...

```
(add-rational (rational 3 4) (rational 2 16) )   ; 56/64
(add-rational (rational 3 4) (rational 4 16) )   ; 64/64
```

$$\frac{3}{4} + \frac{2}{16} = \frac{56}{64}$$ Addition results in a non-reduced fraction...

$$\frac{56 \div 8}{64 \div 8} = \frac{7}{8}$$ ...so we always divide top and bottom by GCD!

Improved rational constructor

```
(define (gcd a b)
    (if (= b 0)
        (abs a)
        (gcd b (modulo a b))))

(define (rational n d)
        (let ((g (if (> d 0)
                     (gcd n d)
                     (- (gcd n d)))))
          (list (/ n g) (/ d g))))
```

Using rationals

User programs can use the rational data abstraction for their own specific needs.

```
; Return 1 + 1/2 + 1/3 + ... + 1/N as a rational number.
(define (nth-harmonic-number n)
    (define (helper rat k)
        (if (= k (+ n 1)) rat
            (helper (add-rational rat (rational 1 k)) (+ k 1))
        )
    )
    (helper (rational 0 1) 1)
)
```

Abstraction barriers

Layers of abstraction

Primitive Representation	`(list n d)` `(car r)` `(car (cdr r))`
Data abstraction	`(rational n d)` `(numer r)` `(denom r)`

	`(add-rational x y)`
	`(mul-rational x y)`
	`(print-rational r)`
	`(are-rationals-equal x y)`

User program	`(nth-harmonic-number n)`

Each layer only uses the layer above it.

Violating abstraction barriers

What's wrong with...

```
(add-rational (list 1 2)  (list 1 4))
```

Violating abstraction barriers

What's wrong with...

```
(add-rational (list 1 2)  (list 1 4))
; Doesn't use constructor!
```

Violating abstraction barriers

What's wrong with...

```
(add-rational (list 1 2)  (list 1 4))
; Doesn't use constructor!
```

```
(define (divide-rationals x y)
    (define new-n (* (car x) (car (cdr y))))
    (define new-d (* (car (cdr x)) (car y)))
    (list new-n new-d)
)
```

Violating abstraction barriers

What's wrong with...

```
(add-rational (list 1 2)  (list 1 4))
; Doesn't use constructor!
```

```
(define (divide-rationals x y)
    (define new-n (* (car x) (car (cdr y))))
    (define new-d (* (car (cdr x)) (car y)))
    (list new-n new-d)
)
; Doesn't use constructor or selectors!
```

Other rational implementations

The `rational` data abstraction could use an entirely different underlying representation.

```
(define (rational n d)
    (define (choose which)
        (if (= which 0) n d)
    )
    choose
)

(define (numer r)
    (r 0)
)

(define (denom r)
    (r 1)
)
```

Rational numbers implementation #2

We could use another abstraction!

```
; Construct a rational number that represents N/D
(define (rational n d)
    (pair n d)
)

; Return the numerator of rational number R.
(define (numer r)
    (first r)
)

; Return the denominator of rational number R.
(define (denom r)
    (second r)
)
```

A tree abstraction

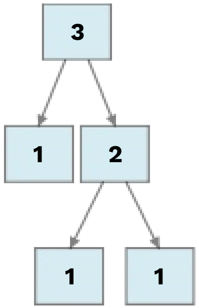
A tree abstraction

We want this constructor and selectors:

`(tree label branches)`	Returns a tree with root `label` and list of `branches`
`(label t)`	Returns the root label of `t`
`(branches t)`	Returns the branches of `t` (each a tree).
`(is-leaf t)`	Returns true if `t` is a leaf node.

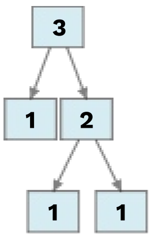
```
(define t
    (tree 3
          (list (tree 1 nil)
                (tree 2 (list (tree 1 nil) (tree 1 nil))))))

(label t)         ; 3
(branches t)      ; ((1) (2 (1) (1)))
(is-leaf t)       ; #f
```


3
1
2
1
1

Tree: Our implementation

```
(define t
    (tree 3
          (list (tree 1 nil)
                (tree 2 (list (tree 1 nil) (tree 1 nil))))))
```



Each tree is stored as a list where first element is label and subsequent elements are branches.

```
(3 (1) (2 (1) (1)))
```

```
(define (tree label branches)
    (cons label branches))

(define (label t) (car t))
```

```
(define (branches t) (cdr t))

(define (is-leaf t) (null? (branches t)))
```

Exercise: Label doubling

Let's implement a Scheme version of the Python function.

```
(define (double tr)
    ; Returns a tree identical to TR, but with all labels doubled.

)
```

```
(define tree1
        (tree 6
            (list (tree 3 (list (tree 1 nil)))
                  (tree 5 nil)
                  (tree 7 (list (tree 8 nil) (tree 9 nil))))))

(expect tree1 (6 (3 (1)) (5) (7 (8) (9))))
(expect (double tree1) (12 (6 (2)) (10) (14 (16) (18))))
```

Exercise: Label doubling (Solution)

Let's implement a Scheme version of the Python function.

```
(define (double tr)
    ; Returns a tree identical to TR, but with all labels doubled.
    (tree (* (label tr) 2) (map double (branches tr)))
)
```

```
(define tree1
        (tree 6
            (list (tree 3 (list (tree 1 nil)))
                  (tree 5 nil)
                  (tree 7 (list (tree 8 nil) (tree 9 nil))))))

(expect tree1 (6 (3 (1)) (5) (7 (8) (9))))
(expect (double tree1) (12 (6 (2)) (10) (14 (16) (18))))
```