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.

```scheme
(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.

```scheme
(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...

\[
\frac{\text{numerator}}{\text{denominator}}
\]

We could use this data abstraction:

<table>
<thead>
<tr>
<th>Constructor</th>
<th>(rational n d)</th>
<th>constructs a new rational number.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selectors</td>
<td>(numer rat)</td>
<td>returns the numerator of the given rational number.</td>
</tr>
<tr>
<td></td>
<td>(denom rat)</td>
<td>returns the denominator of the given rational number.</td>
</tr>
</tbody>
</table>

```
(define quarter (rational 1 4))
(numer quarter) ; 1
(denom quarter) ; 4
```
### Rational number arithmetic

<table>
<thead>
<tr>
<th>Example</th>
<th>General form</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \frac{3}{2} \times \frac{3}{5} = \frac{9}{10} )</td>
<td>( \frac{n_x}{d_x} \times \frac{n_y}{d_y} = \frac{n_x \times n_y}{d_x \times d_y} )</td>
</tr>
<tr>
<td>( \frac{3}{2} + \frac{3}{5} = \frac{21}{10} )</td>
<td>( \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} )</td>
</tr>
</tbody>
</table>
Rational number arithmetic code

We can implement arithmetic using the data abstractions:

<table>
<thead>
<tr>
<th>Implementation</th>
<th>General form</th>
</tr>
</thead>
</table>
| (define (mul_rationals 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 (add_rationals 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)) ) | \[
\frac{n_x}{d_x} + \frac{n_y}{d_y} = \frac{n_x \times d_y + n_y \times d_x}{d_x \times d_y}
\] |
(mul_rationals (rational 3 2) (rational 3 5))
(add_rationals (rational 3 2) (rational 3 5))
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)
  (cons n (cons d nil))
)

; 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))
)
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

<table>
<thead>
<tr>
<th>Primitive Representation</th>
<th>((\text{cons} \ n \ (\text{cons} \ d \ \text{nil})))</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>((\text{car} \ r) \ (\text{car} \ (\text{cdr} \ r)))</td>
</tr>
<tr>
<td>Data abstraction</td>
<td>((\text{rational} \ n \ d))</td>
</tr>
<tr>
<td></td>
<td>((\text{numer} \ r) \ (\text{denom} \ r))</td>
</tr>
<tr>
<td></td>
<td>((\text{add}_\text{rationals} \ x \ y))</td>
</tr>
<tr>
<td></td>
<td>((\text{mul}_\text{rationals} \ x \ y))</td>
</tr>
<tr>
<td></td>
<td>((\text{print}_\text{rational} \ r))</td>
</tr>
<tr>
<td></td>
<td>((\text{are}<em>\text{rationals}</em>\text{equal} \ x \ y))</td>
</tr>
<tr>
<td>User program</td>
<td>((\text{nth}<em>\text{harmonic}</em>\text{number} \ n))</td>
</tr>
</tbody>
</table>

Each layer only uses the layer above it.
Violating abstraction barriers

What's wrong with...

```
(add_rationals (cons 1 (cons 2 nil)) (cons 1 (cons 4 nil)))
```
Violating abstraction barriers

What's wrong with...

(add_rationals (cons 1 (cons 2 nil)) (cons 1 (cons 4 nil)))
; Doesn't use constructors!
Violating abstraction barriers

What's wrong with...

(add_rationals (cons 1 (cons 2 nil)) (cons 1 (cons 4 nil)))
; Doesn't use constructors!

(define (divide_rationals x y)
  (define new_n (* (car x) (car (cdr y))))
  (define new_d (* (car (cdr x)) (car y))
  (cons new_n (cons new_d nil))
)
Violating abstraction barriers

What's wrong with...

```
(add_rationals (cons 1 (cons 2 nil)) (cons 1 (cons 4 nil)))
; Doesn't use constructors!
```

```
(define (divide_rationals x y)
    (define new_n (* (car x) (car (cdr y))))
    (define new_d (* (car (cdr x)) (car y)))
    (cons new_n (cons new_d nil))
)
; Doesn't use selectors!
```
Other rational implementations

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

```scheme
(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!

```scheme
; 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:

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(tree label branches)</td>
<td>Returns a tree with root <code>label</code> and list of <code>branches</code></td>
</tr>
<tr>
<td>(label tr)</td>
<td>Returns the root label of <code>tr</code></td>
</tr>
<tr>
<td>(branches tr)</td>
<td>Returns the branches of <code>tr</code> (each a tree).</td>
</tr>
<tr>
<td>(is-leaf tr)</td>
<td>Returns true if <code>tr</code> is a leaf node.</td>
</tr>
</tbody>
</table>

```
(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
```
Tree: Our implementation

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

```
3
  1 2
  1 1
```

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 tr) (car tr))
```

```
(define (branches tr) (cdr tr))
```
(define (is-leaf tr) (null? (cdr tr)))
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.

```scheme
(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))))
```