Lecture 22: Scheme Data Abstraction

CS 61A - Summer 2024
Raymond Tan

Data Abstraction: Review

Abstraction with a Coffee Machine

We know **what it does**.

1. Add hot water
2. Add coffee beans/pod
3. Press start

We're don't need to know **how it does it**.

"There's a reservoir that holds the water when you pour it into the pot at the start of the coffee-making cycle (on the right in the picture above). At the bottom of the bucket there's a hole, and its role will become obvious in a moment. There's a white tube that leads up from below the reservoir base, carrying the hot water up to the drip area. There is a shower head (on the left-hand side of the picture). Water arrives here from the white hot-water tube and is sprayed over the coffee grounds. In some coffee makers, the water comes out of the hose onto a perforated plastic disc called the drip area and simply falls through the holes into the coffee grounds. The depression on the right-hand side of this figure is the bottom of the bucket. The orange tube on the top picks up the cold water coming down from the hole in the reservoir. The orange tube on the bottom is the hot-water tube (it connects to the white tube that we saw in the previous picture). You can also see the power cord coming in as well. On the left-hand side of the base of the coffee maker is the heating element. This component is comprised of an aluminum extrusion with two parts: a resistive heating element and a tube for water to flow through. The resistive heating element and the aluminum tube heat the water. The resistive heating element is simply a coiled wire, very similar to the filament of a light bulb or the element in an electric toaster that gets hot when you run electricity through it. In a resistive element like this, the coil is embedded in a plaster to make it more rugged. The heating element has two jobs: When you first put the water in the coffee maker, the heating element heats it. Once the coffee is made, the heating element keeps the coffee warm. In the picture above, you can see how the resistive heating element is sandwiched between the warming plate and the aluminum water tube. The resistive heating element presses directly against the underside of the warming plate, and white, heat-conductive grease makes sure the heat transfers efficiently. This grease, by the way, is extremely messy (very hard to get off of your fingers!). You find this grease in all sorts of devices, including stereo amplifiers, power supplies -- pretty much anything that has to dissipate heat. The coffee maker's switch turns power to the heating element on and off. To keep the heating element from overheating, there are also components such as sensors and fuses. In coffee makers, sensors detect if the coil is getting too hot and cut off the current. Then, when it cools down, they turn the current back on. By cycling on and off like this, they keep the coil at an even temperature. Fuses simply cut the power if they sense too high a temperature. They're there for safety reasons, in the event that the main sensor fails. The various sensors and fuses of this coffee maker likely lie in the white sheathing bridging the heating element. The various sensors and fuses of this coffee maker likely lie in the white sheathing bridging the heating element."

Abstraction with a Coffee Machine

- Use / Abstraction
  - Buttons
  - User Interface
  - User Actions

$\longleftarrow$ Abstraction Barrier $\longrightarrow$

- Representation / Hidden Details
  - Internal Mechanisms
  - Complex Processes (heating, brewing, filtering)
  - Electrical Components

Abstraction in Python

- Abstraction hides the inner workings of a function from the users.
- Users only see and interact with the basic implementation, while the complex details are kept hidden.
- Goal: users don’t need to worry about how a function is implemented as long as it acts as expected.

Data Abstraction

- A methodology where functions enforce an abstraction barrier between representation and use.
- An *abstract data type* lets us manipulate compound objects as data.
- Interact with the data abstraction by using:
  - Constructor(s): A function that creates an instance of the data abstraction
  - Selector(s): A function that extracts attribute(s) of the data abstraction

Data Abstraction - Examples

- A car is composed of a gas pedal and brake that make an engine go and stop
- A date can be made from a year, a month, a day
- A linked list can be formed from a first value, a rest (either another link or `Link.empty`)
- A location can be made from a latitude and a longitude

Abstraction Barrier

- We define the barrier between using a data abstraction and how that data abstraction is built as the **abstraction barrier**
- Making assumptions about how a data abstraction is represented is a **violation of the abstraction barrier**
    - Example: When working with the Tree ADT, we must treat each tree object as a **tree**, and **not a list** (the underlying representation)
    - Example: For the Link class, refer to an empty link as **`Link.empty`**, **not an empty tuple** (the underlying representation)
- Reasoning: The creators of the data abstraction can change the underlying representation, but we still want our code to work

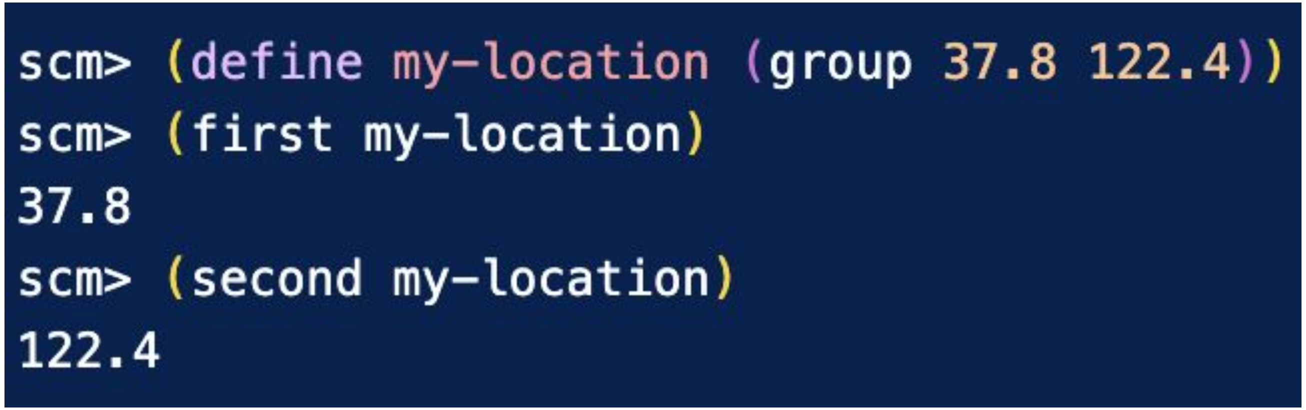
Data Abstraction in Scheme

Group - Scheme Data Abstraction

- Let's come up with a general data abstraction in Scheme - `group`
- `group` will be a general data abstraction that groups two elements together
- Let's look at how it works:
  - Constructor:
    - `(group first-val second-val) -` constructs a new group from 2 arguments
  - Selectors:
    - `(first group-obj) -` accesses and returns the first element of the group
    - `(second group-obj) -` accesses and returns the second element of the group

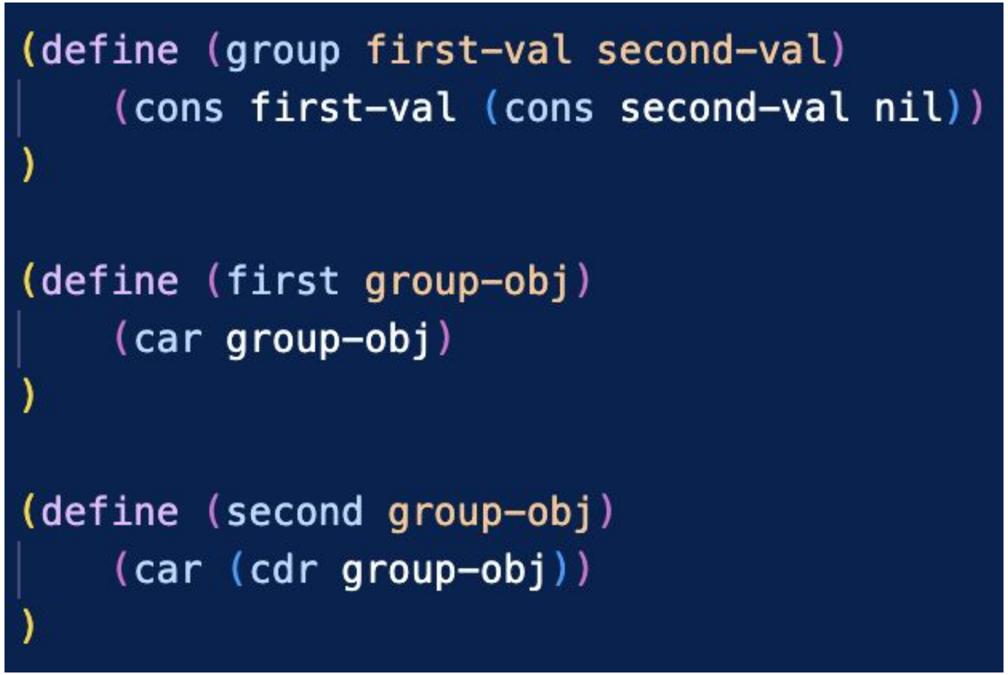
Using group to represent a location

- Let’s use the `group` constructor to represent a location:



```
scm> (define my-location (group 37.8 122.4))
scm> (first my-location)
37.8
scm> (second my-location)
122.4
```

Group - Implementation



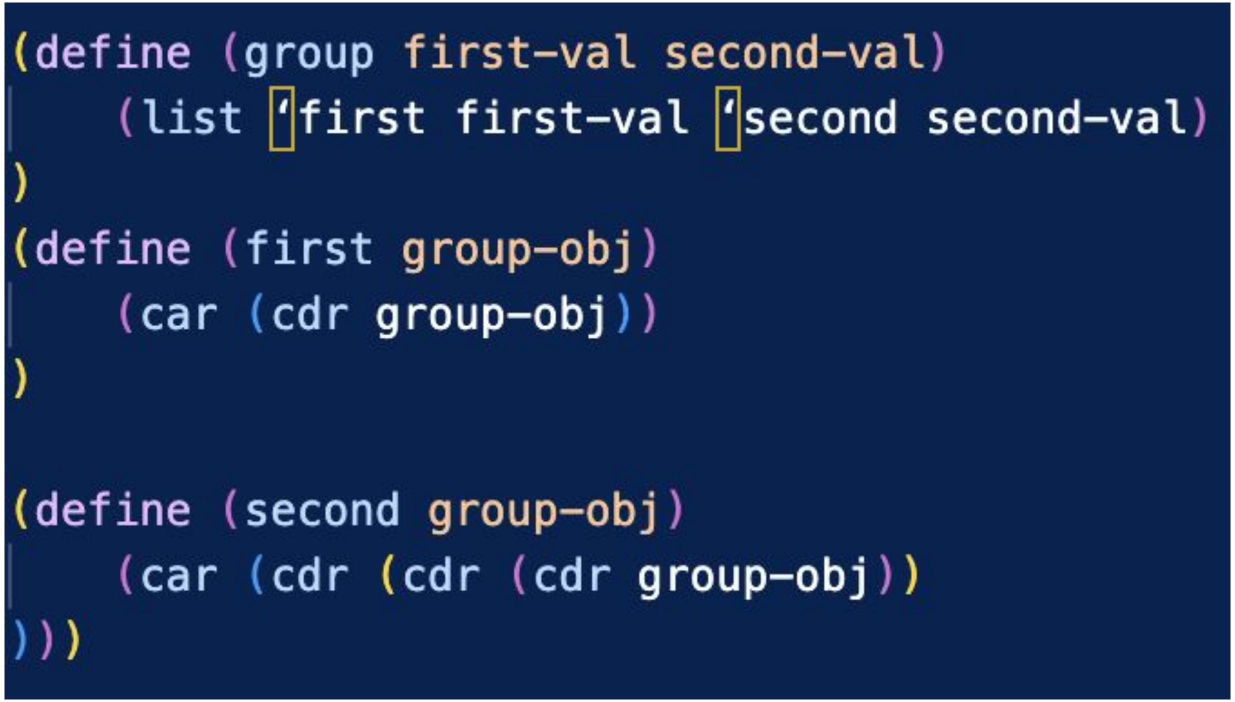
```
(define (group first-val second-val)
    (cons first-val (cons second-val nil))
)

(define (first group-obj)
    (car group-obj)
)

(define (second group-obj)
    (car (cdr group-obj))
)
```

Does it matter how we implemented `group`? Could we have implemented it another way?

Group - Another Implementation



```
(define (group first-val second-val)
    (list ‘first first-val ‘second second-val)
)
(define (first group-obj)
    (car (cdr group-obj))
)

(define (second group-obj)
    (car (cdr (cdr (cdr group-obj))
)))
```

Data Abstraction - Vocab Overview

- **Data Abstraction**: groups compound information into units to be accessed and manipulated.
- **Constructor**: a procedure that creates a data abstraction.
  - Example: `(group first-val second-val)`
- **Selector**: a procedure that accesses and returns some information when a data abstraction is passed into it. Unlike in Python classes, this is a separate procedure and does not have internal access to the data abstraction like self.
  - Example: `(first group-obj)`
- **Documentation**: Information or instructions that describe how a data abstraction work
- **Implementation**: How a data abstraction is coded behind the scenes

Example: rational

Rational Numbers aka fractions

One way fractions can be represented is:

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

So one half = 0.5 = $\frac{1}{2}$

What if I wanted to store exact fractions in scheme?

In Python we might make a class. In Scheme we can make a data abstraction!

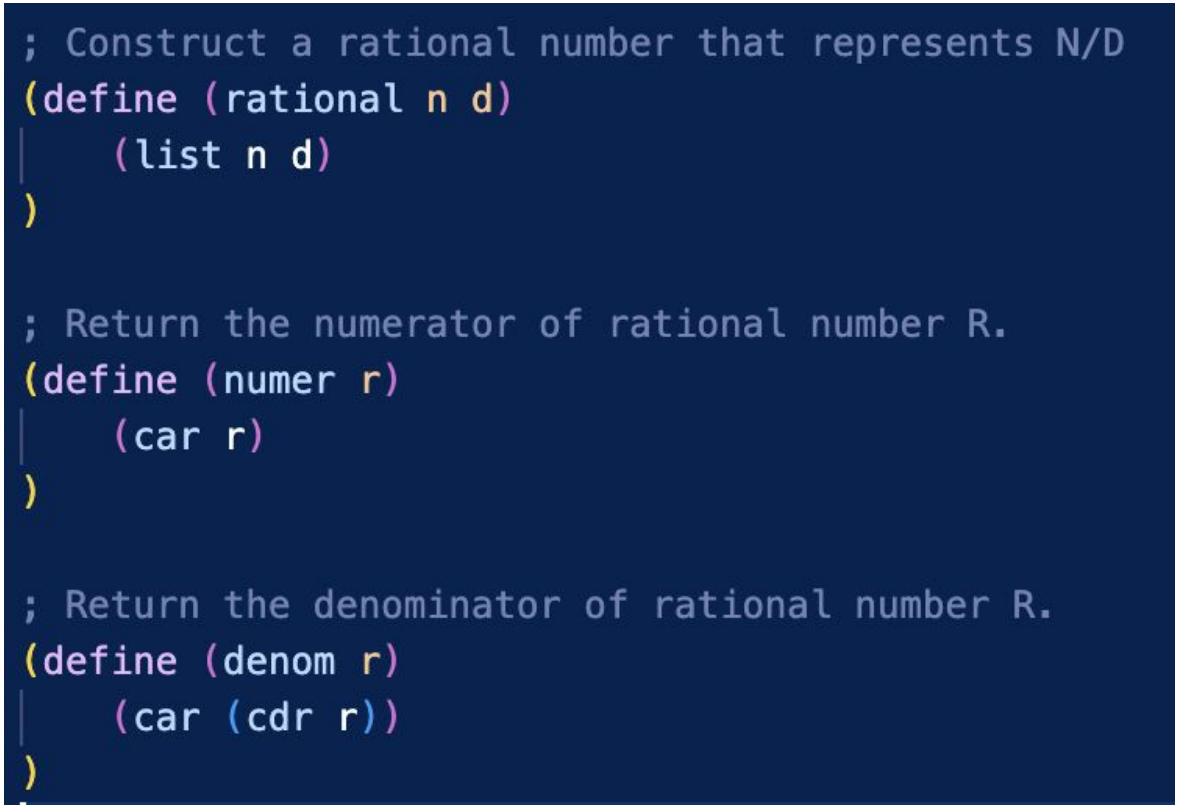
Rational Numbers Documentation

Similar to `group`, we could create some abstraction to access the different parts of the fraction we want to store:

Constructor	`(rational n d)`	Creates a new rational number instance
Selectors	`(numer r)`	Accesses and returns the numerator of the given rational number
	`(denom r)`	Accesses and returns the denominator of the given rational number

The constructor acts like the `__init__` method in a Python class
The selectors act as instance attributes in a Python class, however they are external

Rational Numbers - Implementation

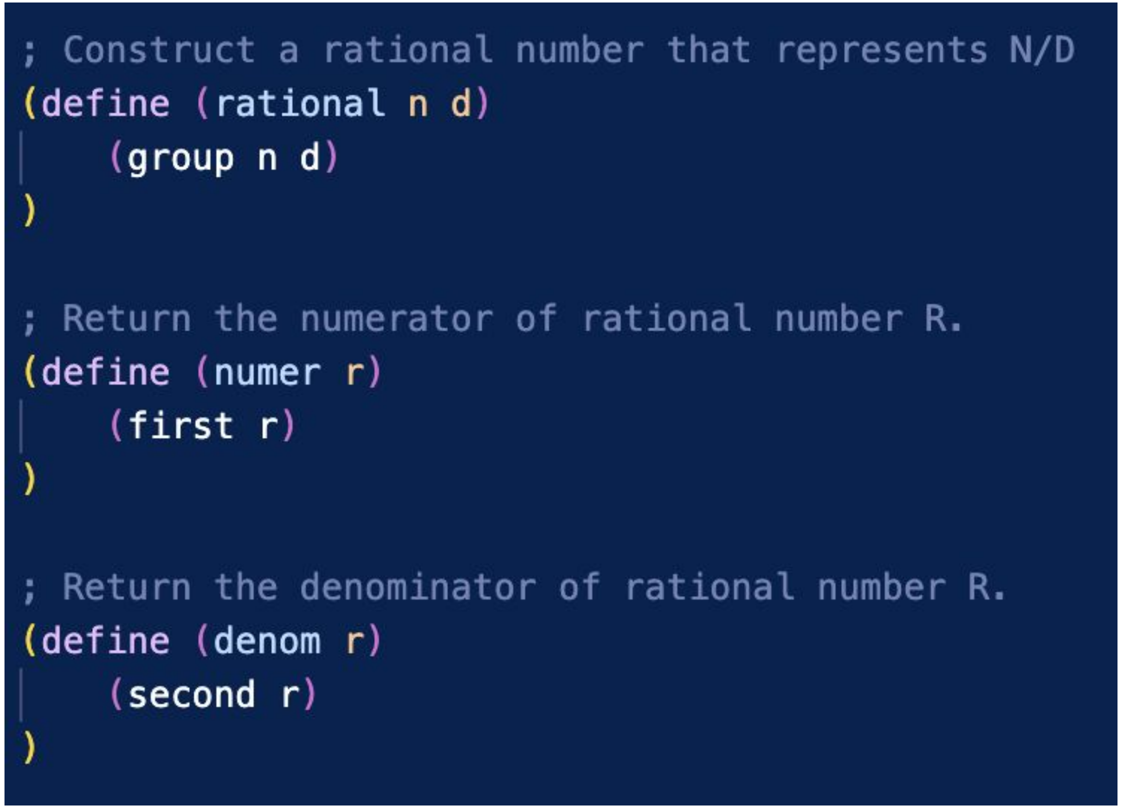


```
; Construct a rational number that represents N/D
(define (rational n d)
    (list n d)
)

; 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))
)
```

Rational Numbers - Implementation #2



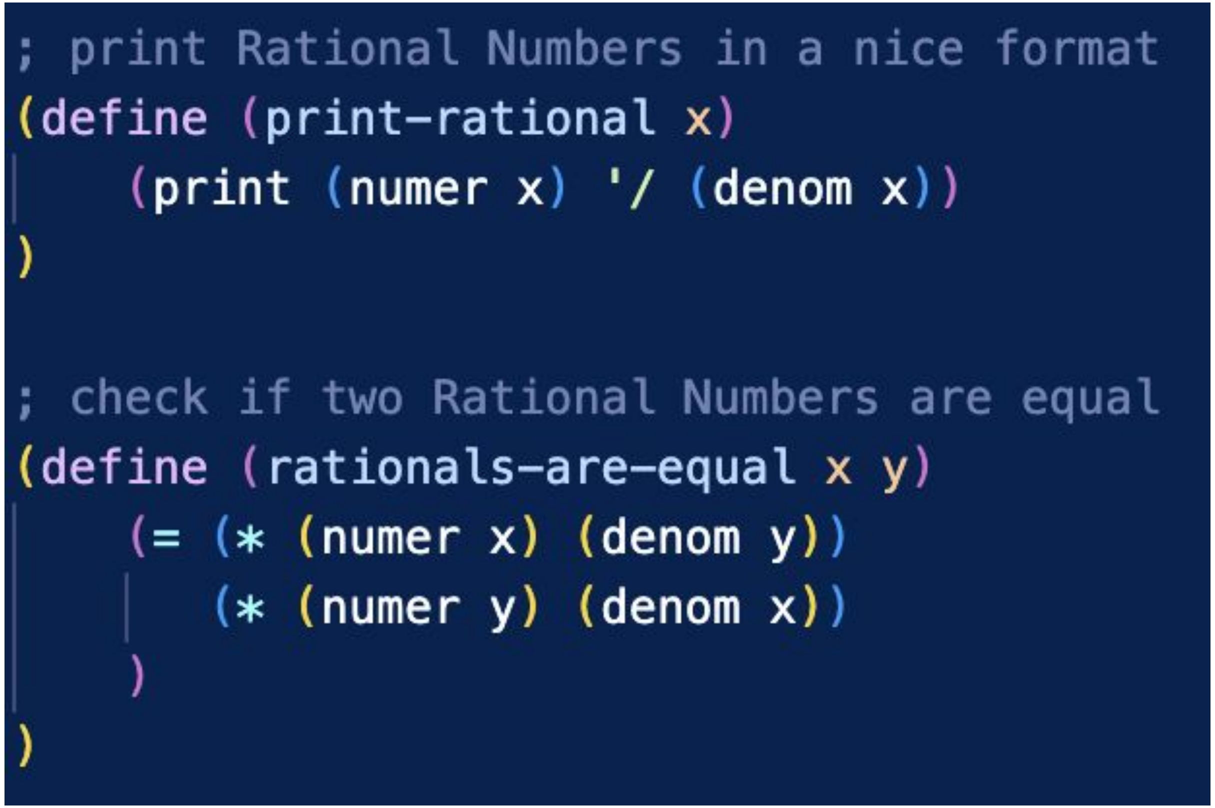
```
; Construct a rational number that represents N/D
(define (rational n d)
    (group n d)
)

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

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

We can use the `group` abstraction that we defined earlier!

Rational Number Utilities



```scheme
; print Rational Numbers in a nice format
(define (print-rational x)
    (print (numer x) '/ (denom x))
)

; check if two Rational Numbers are equal
(define (rationals-are-equal x y)
    (= (* (numer x) (denom y))
       (* (numer y) (denom x))
    )
)
```

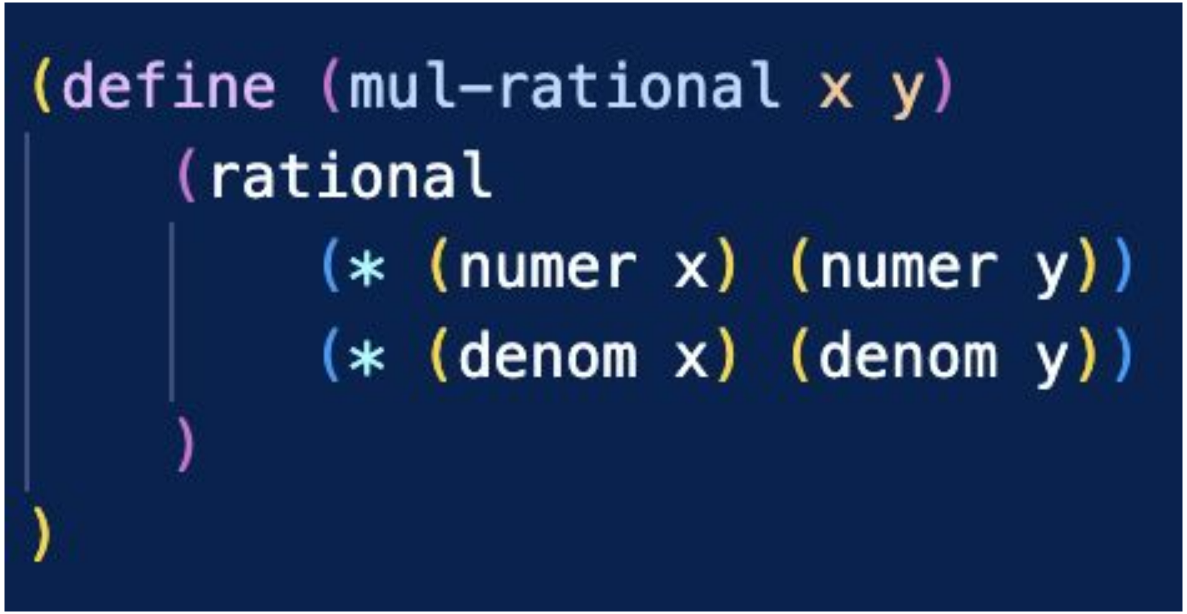
Example: mul-rational

- Problem: Given two rational numbers, a and b, return the multiplication of them.

$$\frac{1}{2} * \frac{3}{4} = \frac{1*3}{2*4} = \frac{3}{8}$$

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

mul-rational: Solution



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

Break

Tree Data Abstraction in Scheme

Tree Abstraction Documentation

What do we need to create and access the parts of a tree?

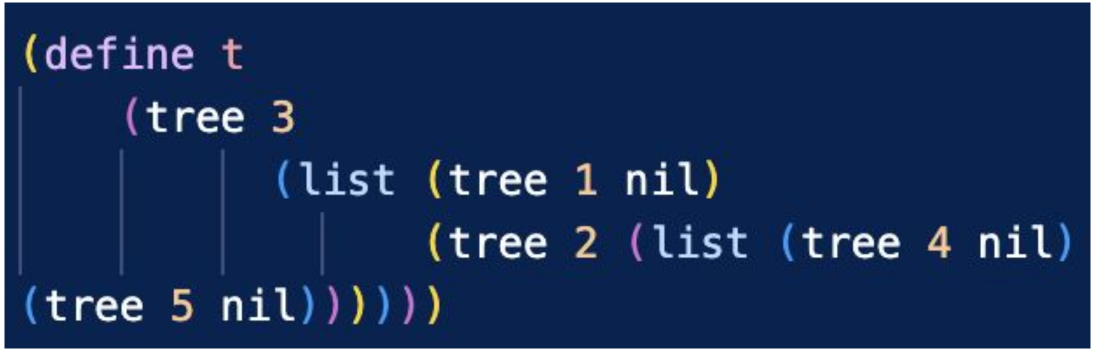
Python	Scheme	

class Tree
Tree(label, branches)
	(tree label branches)	Returns a tree with root label and list of branches

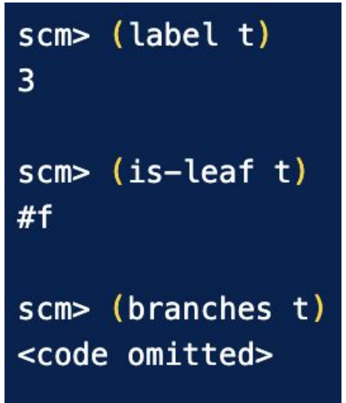
t.label	(label t)	Returns the root label of t
t.branches	(branches t)	Returns the branches of t (trees)
t.is_leaf()	(is-leaf t)	Returns true if t is a leaf node.

Tree Abstraction Documentation

What do we need to create and access the parts of a tree?



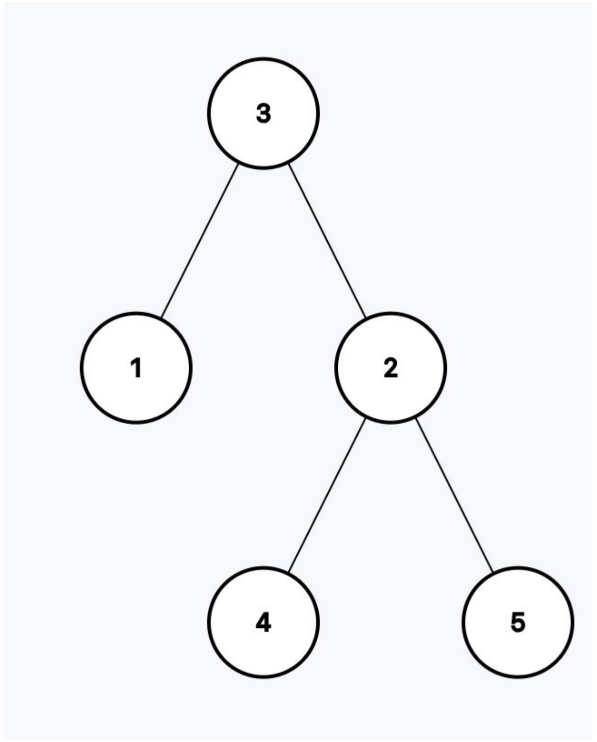
```
(define t
    (tree 3
        (list (tree 1 nil)
            (tree 2 (list (tree 4 nil)
(tree 5 nil))))))
```



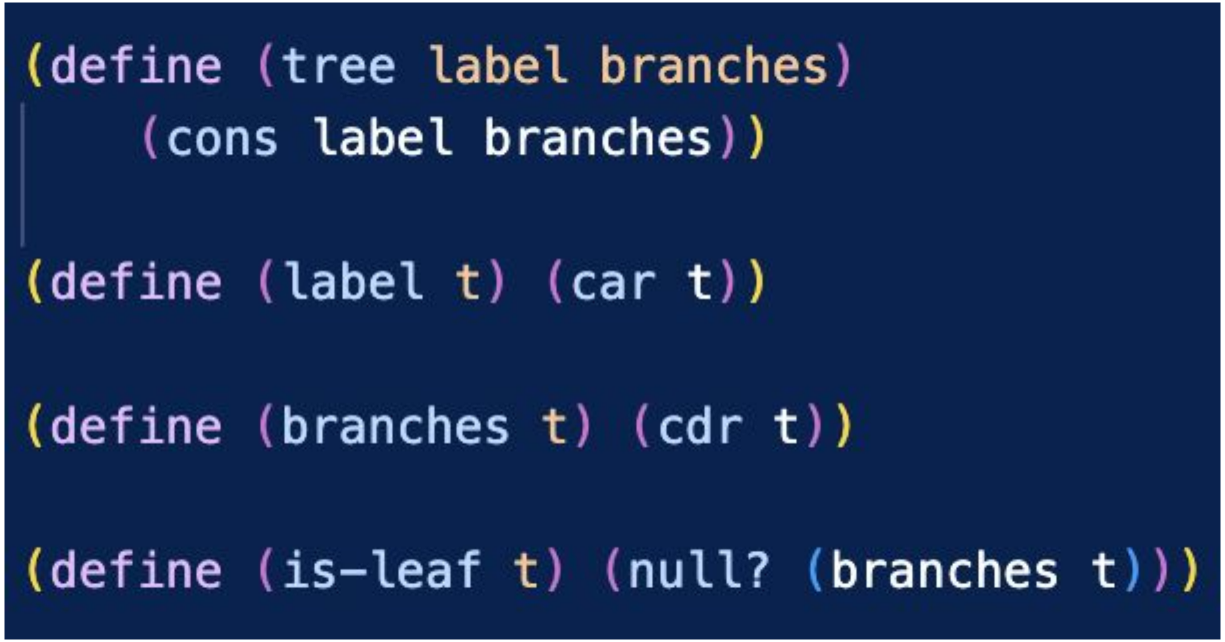
```
scm> (label t)
3

scm> (is-leaf t)
#f

scm> (branches t)
<code omitted>
```



Tree Abstraction - Implementation



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

(define (label t) (car t))

(define (branches t) (cdr t))

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

Remember, this is one of just many ways we can **implement** this abstraction!

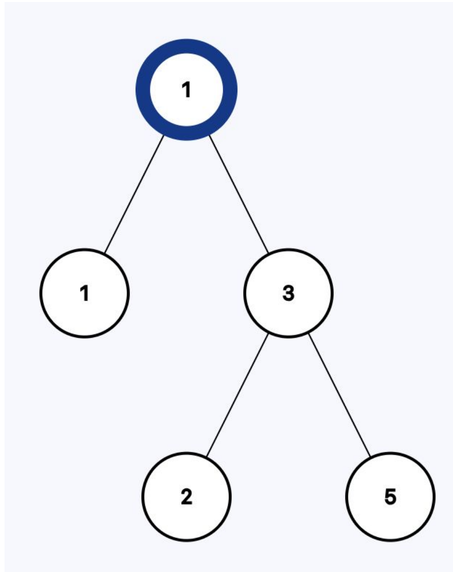
Note on Mutation

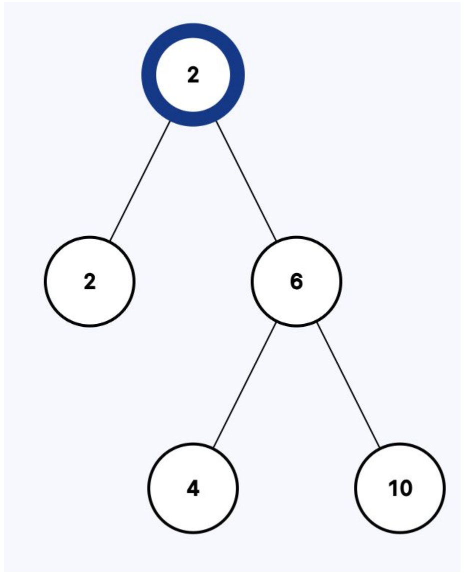
- Comparing this tree implementation to our Python ones, this one is much closer to the Tree ADT, rather than the Tree class
- We will **not** be performing mutative operations on Scheme Tree ADTs!
    - Since there is no formal definition of a class in Scheme

Tree Practice Problems

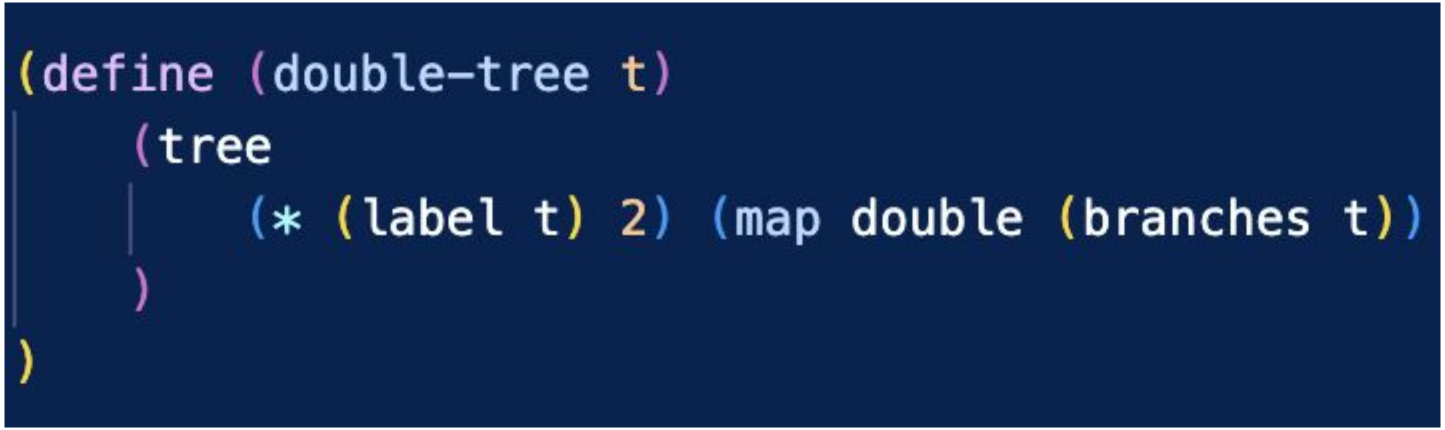
Example: double-labels

- Problem: Given a tree `t` with integer labels, return a new tree where all labels are doubled.





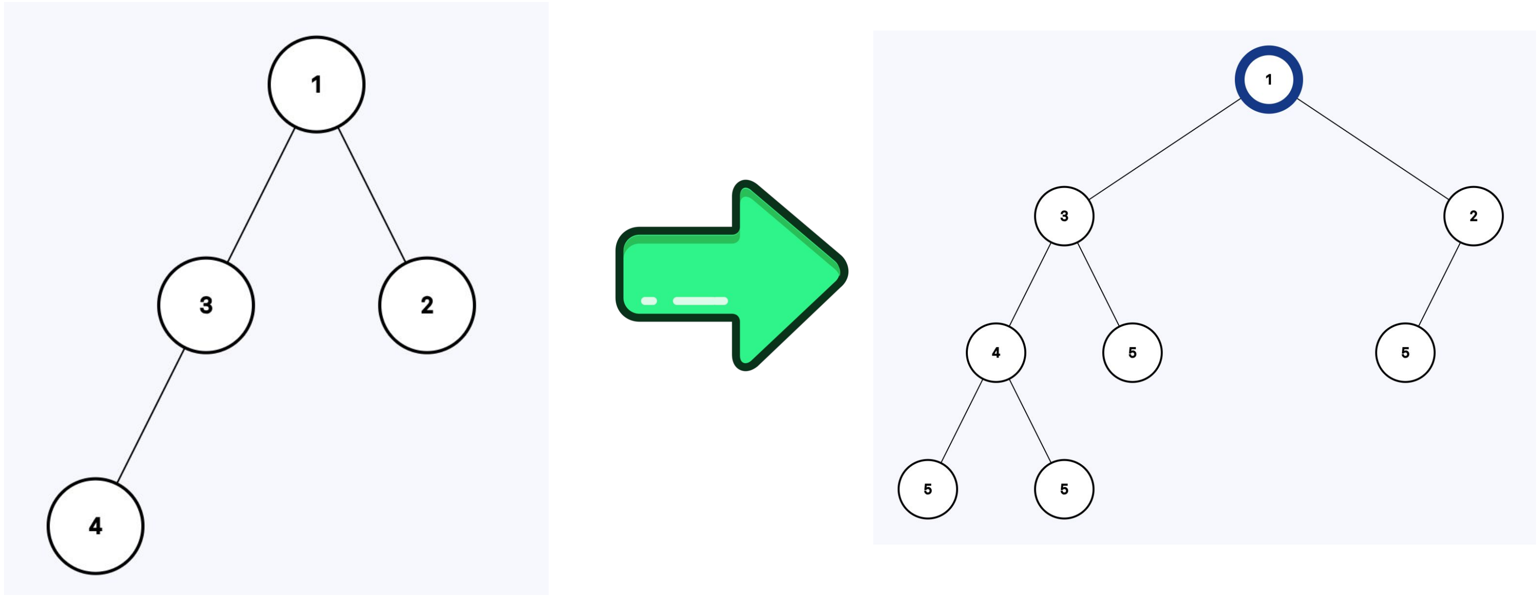
double-labels - Solution



```
(define (double-tree t)
    (tree
        (* (label t) 2) (map double (branches t))
    )
)
```

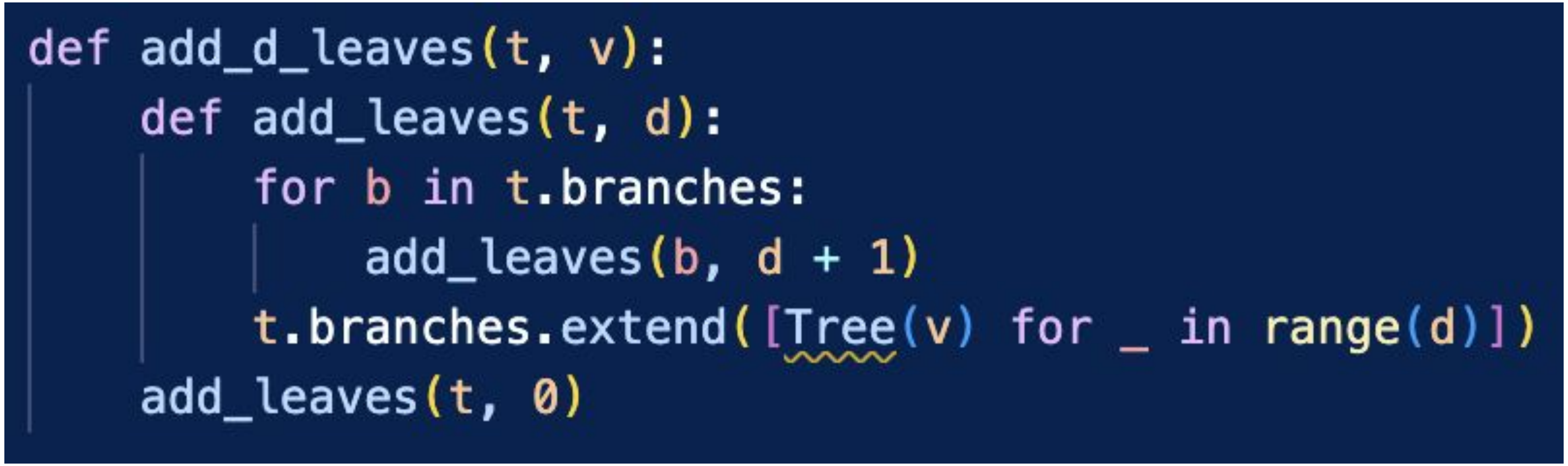
Example: add-d-leaves

- Problem: Given a tree `t`, and a value `v`, return a new tree where each original node has `d` extra leaves added to it. `d` is represented as the depth of a node, where the root has depth 0, the layer below has depth 1, etc



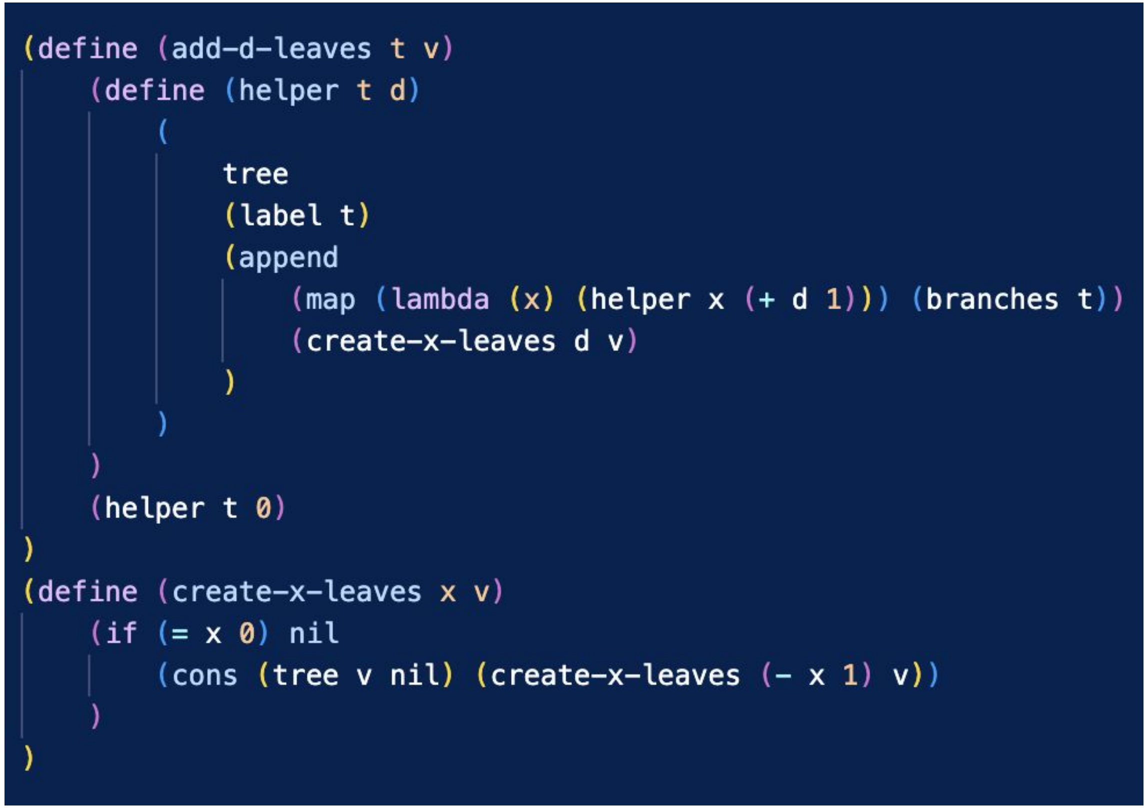
*assuming v=5

add-d-leaves in Python



```python
def add_d_leaves(t, v):
    def add_leaves(t, d):
        for b in t.branches:
            add_leaves(b, d + 1)
        t.branches.extend([Tree(v) for _ in range(d)])
    add_leaves(t, 0)
```

add-d-leaves: Solution



```
(define (add-d-leaves t v)
    (define (helper t d)
        (
            tree
            (label t)
            (append
                (map (lambda (x) (helper x (+ d 1))) (branches t))
                (create-x-leaves d v)
            )
        )
    )
    (helper t 0)
)
(define (create-x-leaves x v)
    (if (= x 0) nil
        (cons (tree v nil) (create-x-leaves (- x 1) v))
    )
)
```

Summary

- Data Abstraction is a methodology where functions enforce an abstraction barrier between representation and use.
- We can enforce the same concepts of data abstraction in Python as we see in Scheme
  - Since there is no formal definition of a class in Scheme, these objects are non-mutable
- Examples in lecture include group, rational, and trees, but we can extend this to anything we'd like to represent!