Lecture 22: Scheme Data Abstraction

July 27th, 2023

Jordan Schwartz

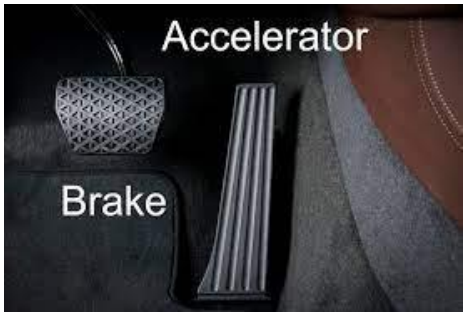
Announcements

- Lab 10 due Today, Thursday 7/27
- Ants due Tomorrow, Friday 7/28
  - Due today, Thursday, 7/27 for EC
- Homework Recovery for HW 4 is due Monday 7/31
- Scheme project will be released Tomorrow, Friday 7/28
- Exam Logistics
  - https://go.cs61a.org/exam-alts
- Mega Section Videos will be delayed today
- No recorded version of this lecture

Data Abstraction

Abstraction

I know cars exist. To drive, I need to know how to put it into drive, where the gas pedal is, and where the brake pedal is. Doesn't really matter what it looks like, as long as they are in the correct place, I can drive the car:



There exist differently engineered mechanisms for car engines and brakes

It doesn't really matter which one the car has though in order to know how to operate it.

The car will go when the gas pedal is pressed, the car will stop when the brakes are pressed.

Data Abstractions

The same concept exists in code. Multiple ideas can be combined into one value:

- A car is composed of 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

In Python we can use data abstractions (like tree) and OOP (like Tree) to store these types of compound values. Scheme doesn't have OOP.

Can you think of any other data types we've discussed so far in this class that may fall under this category?

Data Abstractions

The same concept exists in code. Multiple ideas can be combined into one value:

In Python we can use OOP or dictionaries to store these types of compound values. Scheme doesn't have OOP or dictionary data types.

Data abstractions allow us to group compound values as units. A user doesn't need to worry about how it is implemented as long as it acts as expected

My car can have disc brakes or drum brakes, but as long as the brake pedal slows the car to a stop, it doesn't matter which one the car has!

Group - A Scheme Data Abstraction

If we wanted to group data into “groups” of values, we could use a `group` data abstraction - the concept of a group. Many things can be made with a `group`!

- `(group first-val second-val)` constructs a new group from 2 arguments
  - acts like the `__init__` method in a Python class
- `(first group-obj)` accesses and returns the first value in the given group data abstr.
- `(second group-obj)` accesses and returns the second value in the given group data abstr.
  - selectors act like instance attributes in a Python class, however they are external
  - Methods are internal, inside a Python class, and can access the internal self attribute
  - Scheme selector procedures are external and need a data abreaction passed in

Group - A Scheme Data Abstraction

If we wanted to group data into “groups” of values, we could use a `group` data abstraction with the concept of a group. Many things can be made with a `group`!

```
(define my-location (group 37.8721 122.2578))

scm> (first my-location)
37.8721
scm> (second my-location)
122.2578
```

```
(define common-pets (group ‘dog ‘cat))

scm> (first common-pets)
dog
scm> (second common-pets)
cat
```

Does `group` look similar to any other data types we know of?

Implementation

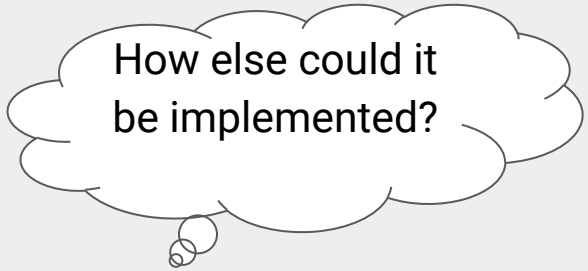
Does it matter how `group` is stored behind the scenes? Only the person who makes the abstraction needs to know and decide how to implement it as long as the constructor and selectors work as expected!

```
(define (group first-val second-val) ; constructor

)

(define (first group-obj) ; selector

)

(define (second group-obj) ; selector

)
```

Implementation

Does it matter how `group` is stored behind the scenes?
Only the person who makes the abstraction needs to know and decide how to implement it!

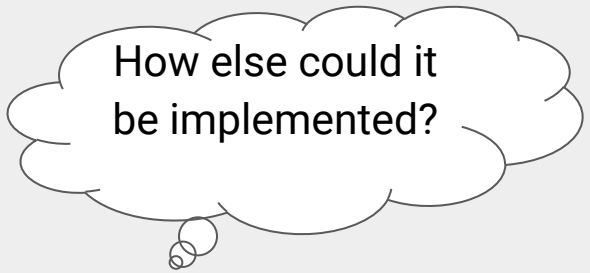
```
(define (group first-val second-val)
    (cons first-val (cons second-val '()))
)
(define (first group-obj)
    (car group-obj)
)
(define (second group-obj)
    (car (cdr group-obj))
)
```



Implementation

Only the person who makes the abstraction needs to know and decide how to implement it!

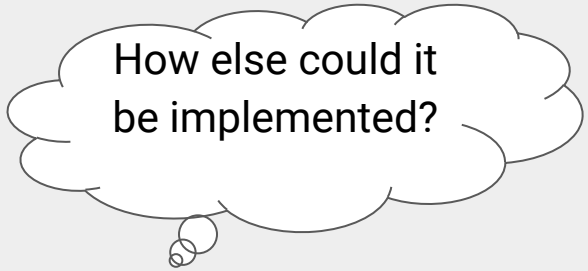
```
(define (group first-val second-val)
    (list 'first first-val 'second second-val)
)
(define (first group-obj)
    (car (cdr group-obj))
)
(define (second group-obj)
    (car (cdr (cdr (cdr group-obj))
)
```



Implementation

You could even go so far as to write it in another language, as long as it behaves as expected. Abstract data types exist in Python as we've seen.

```
def group(first_val, second_val):
    return [first_val, second_val]


def first(group_obj):
    return group_obj[0]


def second(group_obj):
    return group_obj[1]
```



Implementation

It doesn't matter the implementation the developer chooses to use as long as it acts as expected:

```
(define my-location (group 37.8721 122.2578))

scm> (first my-location)

37.8721

scm> (second my-location)

122.2578
```

Vocab Overview

- **Data Abstraction**: groups compound information into units to be accessed and manipulated. Ex: dates, locations, linked lists, trees
- **Constructor**: a procedure that creates a data abstraction. Ex: `(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. Ex: `(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 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 data abstraction or 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 Construction

Now we can store fractions!

```
(define half (rational 1 2))

scm> (numer half)

1

scm> (denom half)

2
```

But what if I want to display them or do arithmetic operations with them?
I'd need to access their data - numerators and denominators

Rational Numbers Utilities

What if I want to view it in a pretty format?

```
(define (print-rational x)
    (print (numer x) '/ (denom x))
)
scm> (print-rational (rational 3 2) )
3 / 2
```

What if I want to check if two rationals are equal? This includes if they are not simplified but have the same value

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

scm> (rationals-are-equal (rational 3 2) (rational 6 4))
#t
```

Rational Number Arithmetic

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

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

$$\frac{1}{2} + \frac{3}{4} = \frac{1*4 + 3*2}{2*4} = \frac{10}{8}$$

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

Rational Number Arithmetic

$$\frac{3}{2} * \frac{3}{5} = \frac{9}{10}$$

$$\frac{3}{2} + \frac{3}{5} = \frac{21}{10}$$

$$\frac{nx}{dx} * \frac{ny}{dy} = \frac{nx*ny}{dx*dy}$$

$$\frac{nx}{dx} + \frac{ny}{dy} = \frac{nx*dy + ny*dx}{dx*dy}$$

General Form

Rational Number Multiplication Code

Using the rational data abstraction from above, let's implement multiplication!

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

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

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

```
scm> (print-rational (mul-rational (rational 1 2) (rational 3 4)))
3 / 8
```

Rational Number Addition Code

Using the rational data abstraction from above, let's implement addition!

```
(define (add-rational x y)
    (rational
        (+
            (* (numer x) (denom y))
            (* (numer y) (denom x))
        )
        (* (denom x) (denom y))
    )
)
```

$$\frac{1}{2} + \frac{3}{4} = \frac{1*4 + 3*2}{2*4} = \frac{10}{8}$$

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

```
scm> (print-rational (add-rational (rational 1 2) (rational 3 4)))
10 / 8
```

Rational Numbers Implementation

One way to store a rational could be with lists in Scheme.

```
; 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))
)
```

Representing Rational Numbers

```
def rational(n, d):
    """Construct a rational number that represents N/D."""
    return [n, d]
```

Construct a list

```
def numer(x):
    """Return the numerator of rational number X."""
    return x[0]

def denom(x):
    """Return the denominator of rational number X."""
    return x[1]
```

Select item from a list

User Programs

I am the developer of `rational` (well, not originally, but in this scenario)! I chose the implementation, the procedures I wanted to include, etc. But what if I wanted to send out `rational` to the world so others could use it too?

I would need to specify what the constructor, the selectors, and procedures are for. Others could then use it without know how it was implemented!

```
; 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)
)
```

Break!

Abstraction Barriers

Abstraction Barriers - They're technically not new!

You've been working with abstraction for a while now – assuming the Tree class and the Linked List class work as intended.

Assuming branches is a list of tree objects

When working with the Tree class you only ever call Tree(label, branches), t.label, t.branches, and is_leaf(t)

When working with the Linked List class you only ever call Link(first, rest), lnk.first, lnk.rest, Link.empty

The same principles apply to Scheme data abstractions – only use the definitions and procedures you are given! Don't assume anything

Layers

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

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

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

Each layer only uses the layer below it

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!
```

We may know how rational is implemented, but we aren't allowed to use that to our advantage.

Other Implementations

You often times won't actually be the developer and may not actually know how a data abstraction works under the hood (eg. `.append()`, `.extend()`, `add` in Python)

Rational could also use an entirely different underlying representation.

```
; Construct a rational number
; that represents N/D
(define (rational n d)
    (define (choose which)
        (if (= which 0) n d)
    )
    choose
)
```

```
; Return the numerator of
; rational number R.
(define (numer r)
    (r 0)
)
; Return the denominator of
; rational number R.
(define (denom r)
    (r 1)
)
```

Other Implementations

It could even use another abstraction!

```
; 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)
)
```

Violating Abstraction Barriers

But what if the user-defined program assumed what the underlying representation was?

Could we, the developers, change our representation?

It would break their code!

Additionally, without using abstraction, your code would get super repetitive.

Tree Abstraction

Tree Abstraction Documentation

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

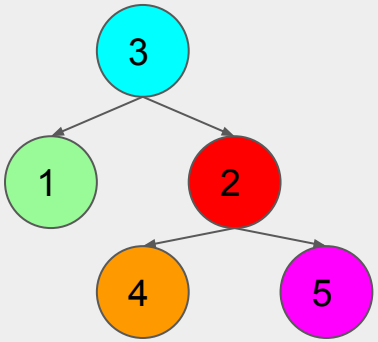
Python Data Abstraction	Python Class	Scheme	
`def tree(label, branches=[])`	`class Tree __init__(self, label, branches=[])`	`(tree label branches)`	Returns a tree with root `label` and list of `branches`
`label(t)`	`t.label`	`(label t)`	Returns the root label of `t`
`branches(t)`	`t.branches`	`(branches t)`	Returns the branches of `t` (trees)
`is_leaf(t)`	`t.is_leaf()`	`(is-leaf t)`	Returns true if `t` is a leaf node.

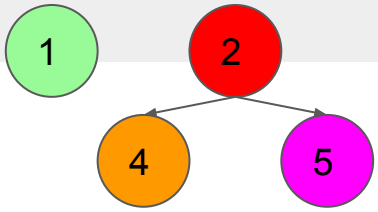
Tree Abstraction Documentation

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

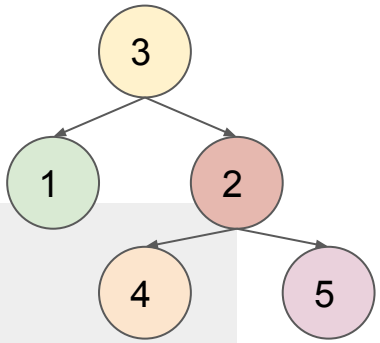
```
(define t
    (tree 3
          (list (tree 1 nil)
                (tree 2 (list (tree 4 nil)
                              (tree 5 nil))))))
```

```
scm> (label t)
3

scm> (is-leaf t)
#f

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





Tree Abstraction Implementation




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

Each tree is stored as a list where the first element is the label of that node and subsequent elements are branches (each branch must also be a tree object!).

```
(3 (1) (2 (4) (5)))
```

```
(3 (1) (2 (4) (5)))
```

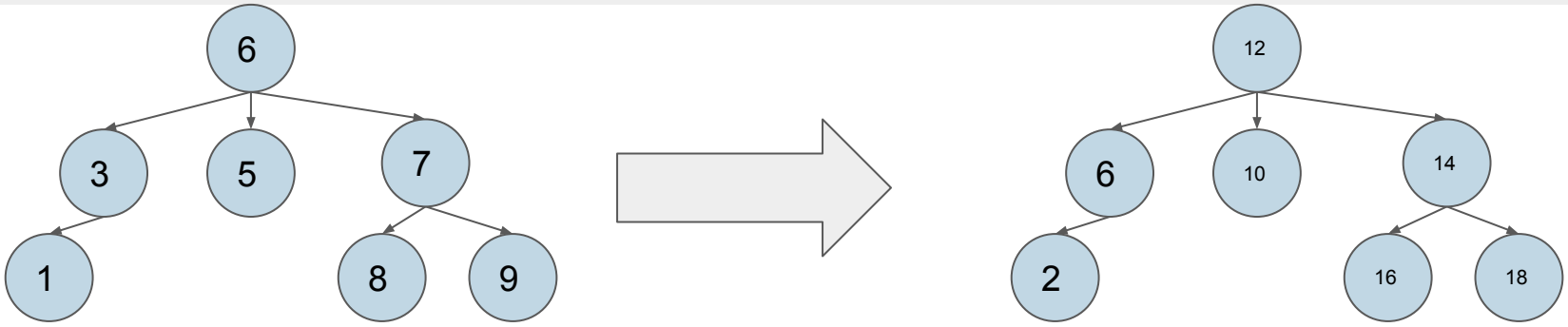
```
(define (tree label branches)
    (cons label branches))

(define (label t) (car t))

(define (branches t) (cdr t))

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

Let’s practice!

Label Doubling

```
(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 <code omitted, left tree below>)
(expect (double tree1) <code omitted, right tree below>)
```



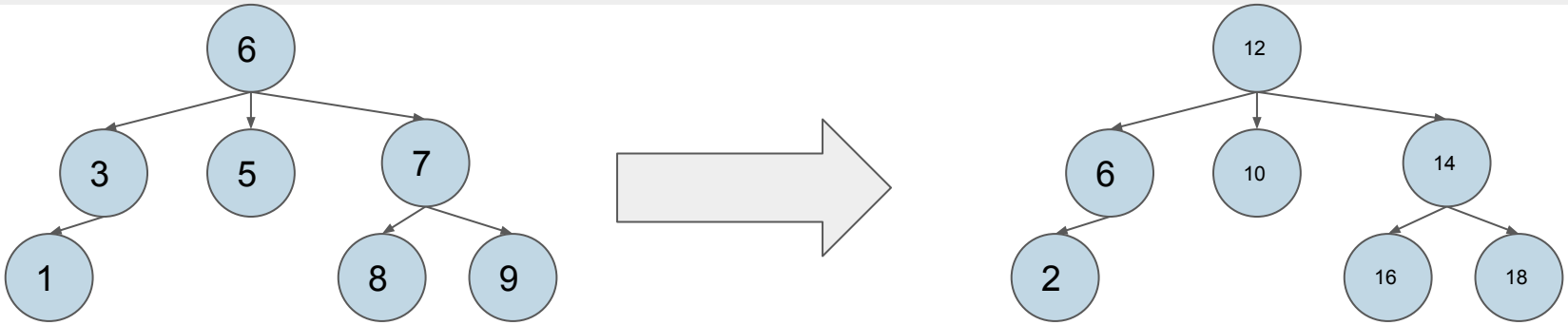
```
def double(t):
    """Returns a tree identical to T, but with all
    labels doubled
    >>> t = tree(1, [tree(2), tree(3)])
    >>> double(t)
    [2, [4], [6]]
    """
    if is_leaf(t):
        return tree(label(t) * 2)
    else:
        return tree(label(t) * 2,
            [double(b) for b in branches(t)])
```

```
def double(t):
    """Returns a tree identical to T, but with all
    labels doubled
    >>> t = tree(1, [tree(2), tree(3)])
    >>> double(t)
    [2, [4], [6]]
    """
    return tree(label(t) * 2,
        [double(b) for b in branches(t)])
```

Label Doubling

```
(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 <code omitted, left tree below>)
(expect (double tree1) <code omitted, right tree below>)
```



More Scheme Practice

Even Subsets

Definition: a *non-empty subset* of a list **s** is a list containing some of the elements of **s**.

(A *non-empty subset* could contain all the elements of s, but not none of them.)

```
;;; Non-empty subsets of integer list s that have an
even sum
;;;
;;; scm> (even-subsets '(3 4 5 7))
;;; ((5 7) (4 5 7) (4) (3 7) (3 5) (3 4 7) (3 4 5))
(define (even-subsets s) ... )
```

A recursive approach: The even subsets of s include...

- all the even subsets of the rest of s
- the first element of s followed by an (even/odd) subset of the rest
- just the first element of s if it is even

(Demo)

Discussion Question: Even Subsets Using Filter

Discussion Question: Complete this implementation of even-subsets

Definition: a *non-empty subset* of a list **s** is a list containing some of the elements of **s**.

(A *non-empty subset* could contain all the elements of s, but not none of them.)

```
;;; non-empty subsets of s
(define (nonempty-subsets s)
 (if (null? s) nil

    (let ((rest  (nonempty-subsets (cdr s))      ))
       (append rest
            (map (lambda (t) (cons (car s) t)) rest)
             (list  (list  (car s) ) ) ) ) ) ) )

;;; non-empty subsets of integer list s that have an even sum
(define (even-subsets s)

 (filter  (lambda (s) (even? (apply + s)))      (nonempty-subsets s)))
```