Rational implementation using functions:
```python
def rational(n, d):
    def select(name):
        if name == 'n':
            return n
        elif name == 'd':
            return d
    return select
```

List comprehensions:
```python
[map expr for <name> in <iter expr> if <filter expr>]
```

A combined expression that evaluates to a list using this evaluation procedure:
1. Add a new frame with the current frame as its parent.
2. Create an empty result list that is the value of the expression.
3. For each element in the iterable value of `<iter expr>`:
   a. Bind `<name>` to that element in the new frame from step 1.
   b. If `<filter expr>` evaluates to a true value, then add the value of `<map expr>` to the result list.

List & dictionary mutation:
```python
>>> a = [10]
>>> b = a
>>> a = [18]
>>> True
>>> a.append(20) # b.append(20)
True
>>> a = b
>>> a
[10, 20]
>>> a = [10, 20]
False
>>> nums = {'X': 10, 'V': 5, 'I': 10}
>>> nums['X']
10
>>> nums['I'] = 1
>>> nums['L'] = 50
>>> nums[3, 9] = 50
>>> nums.get('A', 0)
0
>>> nums.get('W', 0)
5
>>> (x for x in range(3, 6))
(3, 4, 5)
>>> suits = ['coin', 'string', 'myriad']
>>> original_suits = suits
>>> suits.pop()  # 'myriad'
>>> suits
['coin', 'string', 'club']
>>> suits[2] = 'spade'
>>> suits
['coin', 'spade', 'club']
>>> suits[2:2] = ['heart', 'diamond']
>>> suits
['heart', 'diamond', 'spade', 'club']
>>> original_suits
['heart', 'diamond', 'spade', 'club']
```

Identity:
```python
>>> x = y = x
>>> x
5
```

Equality:
```python
>>> x == y
True
```

Types:
```python
>>> isinstance(10, int)
True
>>> isinstance('abc', str)
True
```

Nesting:
```python
>>> def make_withdraw(balance, parent_account):
...     def withdraw(amt):
...         if amt > balance:
...             return 'No funds'
...         return balance - amount
...     return withdraw
...     amount = withdraw(100)
...     amount = withdraw(25)
...     amount = withdraw(25)
...     return amount
... make_withdraw(100, parent_account)
```

Strings as sequences:
```python
>>> city = 'Berkeley'
>>> len(city)
8
>>> 'here' in 'Where's Waldo?'
True
>>> 234 in [1, 2, 3, 4, 5]
False
```

### List & dictionary mutation:
```python
>>> a = [10]  # b = a
>>> a.append(20)  # b.append(20)
>>> a = b
>>> a
[10, 20]
>>> a = [10, 20]  # b = a
False
>>> nums = {'X': 10, 'V': 5, 'I': 10}
>>> nums['X']
10
>>> nums['I'] = 1
>>> nums['L'] = 50
>>> nums[3, 9] = 50
>>> nums.get('A', 0)
0
>>> nums.get('W', 0)
5
>>> (x for x in range(3, 6))
(3, 4, 5)
>>> suits = ['coin', 'string', 'myriad']
>>> original_suits = suits
>>> suits.pop()  # 'myriad'
>>> suits
['coin', 'string', 'club']
>>> suits[2] = 'spade'
>>> suits
['coin', 'spade', 'club']
>>> suits[2:2] = ['heart', 'diamond']
>>> suits
['heart', 'diamond', 'spade', 'club']
>>> original_suits
['heart', 'diamond', 'spade', 'club']
```

Identity:
```python
>>> x = y = x
>>> x
5
```

Equality:
```python
>>> x == y
True
```

Types:
```python
>>> isinstance(10, int)
True
>>> isinstance('abc', str)
True
```

Nesting:
```python
>>> def make_withdraw(balance, parent_account):
...     def withdraw(amt):
...         if amt > balance:
...             return 'No funds'
...         return balance - amount
...     return withdraw
...     amount = withdraw(100)
...     amount = withdraw(25)
...     amount = withdraw(25)
...     return amount
... make_withdraw(100, parent_account)
```

Strings as sequences:
```python
>>> city = 'Berkeley'
>>> len(city)
8
>>> 'here' in 'Where's Waldo?'
True
>>> 234 in [1, 2, 3, 4, 5]
False
```
Tree abstraction:

A tree has a root value and a sequence of branches; each branch is a tree.

```python
def tree(root, branches=None):
    for branch in branches:
        if is_tree(branch):
            return False
    return True

def root(tree):
    return tree[0]

def branches(tree):
    return [branch for branch in branches if not is_leaf(branch)]

def is_leaf(tree):
    return not branches(tree)

def leaves(tree):
    return [leaf for leaf in leaves(tree) if not leaf[0].is_empty]
```

```
Root    5
    ├── 1
    │    └── 0
    └── 3
        ├── 2
        │    └── 1
        └── 1

Leaf has a root value and a sequence of branches; each branch is a tree.
```

```
Leaf

Leaf vs.

Branch

Node

Sub-tree

A new instance is created by calling a constructor:
```python
class Account:
    def __init__(self, account_holder):
        self.balance = 0
        self.account_holder = account_holder
        self.deposit(10)
```
```
>>> a = Account('Jim')
>>> a.balance
0
>>> a.deposit(5)
>>> a.balance
10
```
```
An account instance
```
```python
def is_tree(tree):
    if not tree:
        return False
    for branch in branches(tree):
        if not is_tree(branch):
            return False
    return True
```
```
when a class is called:
```python
1. A new instance of that class is created:
   balance: 0, account holder: 'Jim'
2. The __init__ method of the class is called with the new object as its first argument (named self), along with any additional arguments provided in the call expression.
```