Rational implementation using functions:

```python
def rational(n, d):
    if name == 'n':
        return n
    elif name == 'd':
        return d

return select
```

### Lists:

- **digits** = [1, 8, 2, 8]
- **len(digits)** = 4
- **digits[3]**
- **[2, 7] + digits**
- **pairs** = [(10, 20), (30, 40)]
- **pairs[1]**
- **pairs[0] = 30**

Executing a for statement:

```python
for <name> in <expression>:
```

1. Evaluate the header `<expression>`, which must yield an iterable value (a list, tuple, iterator, etc.).
2. For each element in that sequence, in order:
   - **A.** Bind `<name>` to that element in the current frame.
   - **B.** Execute the `<suite>`

Unpacking in a for statement:

```python
pairs = [(1, 2), [2, 3], [3, 4, 5], [4, 5, 6, 7], [5, 6, 7, 8, 9, 10], [6, 7, 8, 9, 10, 11]]
```

Length: ending value - starting value

Element selection: starting value + index

```python
list(range(-2, 2))
```

Membership:

```python
digits = [1, 8, 2, 8]
digits[2] in digits
''
not in digits
```

Identity:

```python
exp1 is exp2
```

### List comprehensions:

```python
[<map exp for <name> in <iter exp> if <filter exp>]
```

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 exp>`:
   - **A.** Bind `<name>` to that element in the new frame from step 1.
   - **B.** If `<filter exp>` evaluates to a true value, then add the value of `<map exp>` to the result list.

The result of calling `repr` on a value is what Python prints using the `__repr__` method:

```python
repr(123)
```

The result of calling `str` on a value is what Python prints using the `__str__` method:

```python
str(123)
```

Type dispatching:

- **functions**
- **expressions**

### Type coercion:

Look up a function for converting one type to another, then apply a type-specific implementation.

```python
def some_function(x):
    return x
```

You can copy a list by calling the list constructor or slicing the list from the beginning to the end.

```python
list(range(10))
```

### Lists & dictionaries:

- **nums = {'x': 10, 'y': 5, 'z': 18, 'w': 'A'}
- **len(nums)**
- **nums['x']**
- **nums['y']**

Type coercion:

```python
int('123')
```

False values:

- **bool(False)**
- **not False**
- **None**
- **not None**

All other values are true values.

```python
not None
```

### Global frame:

```python
make_withdraw(amount)
```

### Status:

- **No nonlocal statement**
- **x = 2**
- **y**

### Effect:

- **Create a new binding from name "x" to number 2**
- **Re-bind name "x" to object 2 in the first frame of the current environment**
- **Re-bind "x" to 2 in the first non-local frame of the current environment**
- **SyntacticError: no binding for nonlocal "x" found**
- **SyntacticError: name 'x' is parameter and nonlocal name 'x' also bound locally**

### Lists:

- **digits**
- **list(digits)**
- **len(digits)**
- **digits[3]**
- **[2, 7] + digits**
- **pairs**
- **len(pairs)**
- **pairs[1]**
- **pairs[0] = 30**

### Functions:

- **def**: A function is a first-class object
- **return**: A function returns a value
- **def**: Function definition
- **return**: Function return
- **pass**: Function pass
- **del**: Function del
- **None**: Function None

### Slicing:

- **List slicing**
- **tuple slicing**
- **iterator slicing**
- **list comprehension**

### Equality:

- **==**: Equality comparison
- **!=**: Not equal comparison
- **<**: Less than comparison
- **<=**: Less than or equal comparison
- **>**: Greater than comparison
- **>=**: Greater than or equal comparison

### Operators:

- **+**: Addition operator
- **-**: Subtraction operator
- *****: Multiplication operator
- **/**: Division operator
- **%**: Modulo operator
- **//**: Floor division operator
- ******: Exponentiation operator
- **==**: Equality operator
- **!=**: Not equal operator
- **<**: Less than operator
- **<=**: Less than or equal operator
- **>**: Greater than operator
- **>=**: Greater than or equal operator
- **and**: Logical and operator
- **or**: Logical or operator
- **not**: Logical not operator
- **in**: Membership operator
- **is**: Identity operator

### Boolean values:

- **True**: Boolean value
- **False**: Boolean value

### Control flow:

- **if**: Conditional statement
- **elif**: Conditional statement
- **else**: Conditional statement
- **for**: Loop statement
- **while**: Loop statement
- **break**: Loop break
- **continue**: Loop continue
- **return**: Function return
- **yield**: Generator expression

### Iterators:

- **iter(iterable)**: Iteration object
- **next(iterator)**: Next value
- **forall(iterator)**: Boolean value

### Iterables:

- **list(iterable)**: List
- **tuple(iterable)**: Tuple
- **set(iterable)**: Set
- **frozenset(iterable)**: Frozen set
- **dict(iterable)**: Dictionary

### Container objects:

- **list**: A sequence of elements
- **tuple**: A tuple of elements
- **set**: A set of elements
- **frozenset**: A frozenset of elements
- **dict**: A dictionary of key-value pairs

### Iteration:

- **for in iterable**: Iteration over elements in iterable
- **while condition**: Iteration while condition

### Comprehensions:

- **list comprehension**: List comprehension
- **set comprehension**: Set comprehension
- **dict comprehension**: Dictionary comprehension

### Generators:

- **generator expression**: Generator expression
- **generator object**: Generator object

### Lambda:

- **lambda**
- **map(, , , )**: Map function
- **filter(, , , )**: Filter function
- **reduce(, , )**: Reduce function

### Functions:

- **def**: Function definition
- **return**: Function return
- **yield**: Generator expression
- **lambda**: Lambda expression

### Decorators:

- **@**
- **decorator**: Decorator

### Objects:

- **object**: First-class object
- **method**: Method
- **property**: Property
- **raw**: Raw object
- **proxy**: Proxy object

### Classes:

- **class**: Class definition
- **instance**: Instance of class
- **method**: Method of instance

### Exceptions:

- **Exception**: Base exception class
- **BaseException**: Base exception class
- **Exception**
- **BaseException**
- **ValueError**: Value error exception
- **TypeError**: Type error exception
- **IOError**: IO error exception
- **RuntimeError**: Runtime error exception
- **SyntaxError**: Syntax error exception
- **Indeterminate**: Indeterminate error exception
Recursive cases are evaluated

Conditional statements check for

The

and

count_partitions(2, 4)

Use at least one 4

Recursive description:

• A tree has a root label
• Each branch is a tree
• A tree with zero branches is called a leaf

Relative description:

• Each location is a node
• Each node has a label
• One node can be the parent/child of another

def tree(label, branches=[]):
    if branches:
        for branch in branches:
            assert is_tree(branch)
    return [label] + list(branches)
def label(tree):
    return tree[0]

def branches(tree):
    return tree[1:]

def is_tree(tree):
    if not isinstance(tree, list):
        return False
    if len(tree) < 2:
        return False
    if not is_tree(tree[0]):
        return False
    for branch in branches(tree):
        if not is_tree(branch):
            return False

self should always be bound to an instance of
the Account class or a subclass of Account

Recursive decomposition: finding simpler instances of a problem.
- E.g., count_partitions(6, 4)
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
- Solve two simpler problems:
  - count_partitions(6, 3)
  - count_partitions(6, 2)
- Tree recursion often involves
  exploring different choices.

An account instance

cs 61a midterm 2 study guide — page 2

Recursive description:

• A tree has a root label
• Each branch is a tree
• A tree with zero branches is called a leaf

Relative description:

• Each location is a node
• Each node has a label
• One node can be the parent/child of another

def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [label] + list(branches)
def label(tree):
    return tree[0]

def branches(tree):
    return tree[1:]

def is_tree(tree):
    if not isinstance(tree, list):
        return False
    if len(tree) < 2:
        return False
    if not is_tree(tree[0]):
        return False
    for branch in branches(tree):
        if not is_tree(branch):
            return False

self should always be bound to an instance of
the Account class or a subclass of Account

Recursive decomposition: finding simpler instances of a problem.
- E.g., count_partitions(6, 4)
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
- Solve two simpler problems:
  - count_partitions(6, 3)
  - count_partitions(6, 2)
- Tree recursion often involves
  exploring different choices.