The interface for sets:
- Membership testing: Is a value an element of a set?
- Adjunction: Return a set with all elements in s and a value v.
- Union: Return a set with all elements in set1 or set2.
- Intersection: Return a set with any elements in set1 and set2.

<table>
<thead>
<tr>
<th>Union</th>
<th>Intersection</th>
<th>Adjunction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2</td>
<td>1 3</td>
<td>1 2</td>
</tr>
<tr>
<td>4 3</td>
<td>1 3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 3</td>
<td></td>
</tr>
</tbody>
</table>

Proposal 1: A set is represented by a recursive list that contains no duplicate items.

Proposal 2: A set is represented by a recursive list with unique elements ordered from least to greatest.

Proposal 3: A set is represented as a Tree. Each entry is:
- Larger than all entries in its left branch and
- Smaller than all entries in its right branch

If 9 is in the set, it is somewhere in this branch.

Exceptions are raised with a raise statement.

```python
raise <expression>
```

<expression> must evaluate to an exception instance or class.

Exceptions are constructed like any other object; they are just instances of classes that inherit from BaseException.

```python
try:
    <try suite>
except <exception class> as <name>:
    <exception suite>
```

The <try suite> is executed first;
If, during the course of executing the <try suite>, an exception is raised that is not handled otherwise, and If the class of the exception inherits from <exception class>, then
The <exception suite> is executed, with <name> bound to the exception.

```python
class Stream(object):
    '''A lazily computed recursive list.'''
    class empty(object):
        def __repr__(self):
            return 'Stream.empty'
        empty = empty()

    def __init__(self, first, compute_rest=lambda Stream.empty):
        assert callable(compute_rest), 'compute_rest must be callable.'
        self.first = first
        self.compute_rest = compute_rest
        self.rest = None

@property
    def __next__(self):
        """Please don't reference directly"""
        return self.rest

    def __iter__(self):
        yield self.first
        return Stream(self.rest, lambda Stream(): Stream.empty())

    def __len__(self): return len(self.first) + len(self.rest)

    def __repr__(self):
        return 'Stream(%s, %s)' % (self.first, self.rest)

class integer_stream(Stream):
    def compute_rest(self):
        return integer_stream(first=x, compute_rest=lambda Stream.empty(): Stream.empty())

    def __iter__(self):
        yield x
        return Stream(self.rest, lambda Stream(): Stream.empty())

    def filter_stream(self, fn, s):
        if s is Stream.empty:
            return s
        def compute_rest():
            return filter_stream(fn, s.rest)
        self.first = first
        self.compute_rest = compute_rest
        self.rest = None
        return self

    def map_stream(self, fn, s):
        if s is Stream.empty:
            return s
        def compute_rest():
            return self
        self.first = first
        self.compute_rest = compute_rest
        self.rest = None
        return self

    def primes(self, pos_stream):
        def not_divisible(x):
            return primes(pos_stream)
        def compute_rest():
            return self
        return primes(self.filter_stream(not_divisible, pos_stream))

    def __next__(self):
        return next(self)

    def __len__(self):
        return len(self.first) + len(self.rest)

    def __iter__(self):
        yield NEXT(self)
        return Stream(self.rest, lambda Stream(): Stream.empty())

def rational_stream(n, s):
    if n is Stream.empty:
        return s
    def compute_rest():
        return rational_stream(n, s.rest)
    self.first = first
    self.compute_rest = compute_rest
    self.rest = None
    return self

    def __iter__(self):
        yield x
        return Stream(self.rest, lambda Stream(): Stream.empty())

    def __len__(self):
        return len(self.first) + len(self.rest)

    def __iter__(self):
        yield NEXT(self)
        return Stream(self.rest, lambda Stream(): Stream.empty())

A simple fact expression in the Logic language declares a relation to be true.

Language Syntax:
- A relation is a Scheme list.
- A fact expression is a Scheme list of relations.

```python
logict (fact (parent delano herbert))
logict (fact (parent abraham barack))
logict (fact (parent abraham clinton))
logict (fact (parent fillmore delano))
logict (fact (parent fillmore grover))
logict (fact (parent eisenhower fillmore))
```

Relations can contain relations in addition to atoms.

```python
logict (fact (dog (name abraham) (color white)))
logict (fact (dog (name barack) (color tan)))
logict (fact (dog (name clinton) (color white)))
logict (fact (dog (name delano) (color white)))
logict (fact (dog (name eisenhower) (color tan)))
logict (fact (dog (name fillmore) (color brown)))
logict (fact (dog (name grover) (color tan)))
logict (fact (dog (name herbert) (color brown)))
```

Variables can refer to atoms or relations in queries.

```python
logict (query (parent barack ?child))
Success!
child: barack
child: clinton
```

```python
logict (query (dog (name clinton) ?color))
Success!
color: white
```

```python
logict (query (dog (name clinton) ?info))
Success!
info: (color white)
```

A fact can include multiple relations and variables as well:

```python
logict (query conclusion hypothesis, hypothesis, ... hypothesis)
```

Means <conclusion> is true if all <hypothesis> are true.

```python
logict (fact (child ?p) (parent ?p ?c))
logict (query child herbert delano)
Success!

logict (query (child eisenhower clinton))
Failure.
```

The Logic interpreter searches in the space of relations for each query to find a satisfying assignment.

{parent delano herbert : (1), a simple fact}
{ancestor delano herbert : (2), from the 1st ancestor fact
{parent fillmore delano : (3), a simple fact

{ancestor fillmore herbert : (4), from (2), (3), & the 2nd ancestor fact

Two lists append to form a third list if:
- The first list is empty and the second and third are the same
- The rest of 1 and 2 append to form the rest of 3

```python
logict (fact (append-to-form ?x ?y))
logict (fact (append-to-form ?a . ?r) ?y (7a . 2z))
```

```python
class Letters(object):
    '''An iterator over letters.'''
    class empty(object):
        def __repr__(self):
            return 'Letters.empty'
        empty = empty()

    def __init__(self, s):
        self.current = 'a'
        self.next = self.next

    def __len__(self):
        return len(s)

    def __iter__(self):
        yield self.current
        self.current = chr(ord(self.current) + 1)
        return self.next

    def __next__(self):
        yield self.current
        self.current = chr(ord(self.current) + 1)
        self.next = self.next

    def __repr__(self):
        return 'Letters(%s)' % self

    def __len__(self):
        return len(self.current) + len(self.next)

    def __iter__(self):
        yield NEXT(self)
        return Stream(self.rest, lambda Stream(): Stream.empty())

    def __next__(self):
        return NEXT(self)

    def __len__(self):
        return len(self.current) + len(self.next)

    def __iter__(self):
        yield NEXT(self)
        return Stream(self.rest, lambda Stream(): Stream.empty())

    def __next__(self):
        return NEXT(self)

    def __len__(self):
        return len(self.current) + len(self.next)

    def __iter__(self):
        yield NEXT(self)
        return Stream(self.rest, lambda Stream(): Stream.empty())

    def __next__(self):
        return NEXT(self)

    def __len__(self):
        return len(self.current) + len(self.next)

    def __iter__(self):
        yield NEXT(self)
        return Stream(self.rest, lambda Stream(): Stream.empty())

letters = Letters()
 letters.next()  # 'a'
 letters.next()  # 'b'
 letters.next()  # 'c'
 letters.next()  # 'd'

for x in Letters():
    print(x)
```

```python
class LetterIterable(object):
    '''An iterable over letters.'''
    def __init__(self, s):
        self.current = 'a'
        self.next = self.next

def __len__(self):
    return len(s)

def __iter__(self):
    yield self.current
    self.current = chr(ord(self.current) + 1)
    self.next = self.next

def __next__(self):
    yield self.current
    self.current = chr(ord(self.current) + 1)
    self.next = self.next

def __repr__(self):
    return 'LetterIterable(%s)' % self

def __len__(self):
    return len(self.current) + len(self.next)

def __iter__(self):
    yield NEXT(self)
    return Stream(self.rest, lambda Stream(): Stream.empty())

def __next__(self):
    return NEXT(self)

def __len__(self):
    return len(self.current) + len(self.next)

def __iter__(self):
    yield NEXT(self)
    return Stream(self.rest, lambda Stream(): Stream.empty())

letters = Letters()
 letters.next()  # 'a'
 letters.next()  # 'b'
 letters.next()  # 'c'
 letters.next()  # 'd'

for x in Letters():
    print(x)
```
CS 61A Final Exam Study Guide – Page 2

Scheme programs consist of expressions, which can be:
- Primitive expressions: 2, 3.3, true, quotient ...
- Combinations: (quotient 10 2) (not true) 
  Numbers are self-evaluating; symbols are bound to values.
  Call expressions have an operator and 0 or more operands.

A combination that is not a call expression is a special form:
  • If expression: (if <predicate> <consequent> <alternative>)
  • Binding names: (define <name> <expression>)
  • New procedures: (define <name> [<formal-parameters>] <body>)

Lambda expressions evaluate to anonymous functions.
\[\text{(define plus4 (lambda (x) (+ x 4)))}\]
Lambda expressions evaluate to anonymous functions.

A Scheme value can be a combination or an atom.

A. For each element in that sequence, in order:
  - Evaluate the header
  - Check for procedures
  - Evaluate arguments

Lambda expressions evaluate to anonymous functions.

A Scheme list is written as elements in parentheses:

```
(define (abs x)
  (if (< x 0)
      (- x)
      x))
```

Lambda expressions evaluate to anonymous functions.

A Scheme list is written as elements in parentheses:

```
A number or a Pair with an operator as its first element
```

Lambda expressions evaluate to anonymous functions.

A Scheme list is written as elements in parentheses:

```
  * (if (null? s) '() (cdr s))
          (+ 1 (- 23) (+ 4.56))
```

Lambda expressions evaluate to anonymous functions.

A Scheme list is written as elements in parentheses:

```
A number or a Pair with an operator as its first element
```

Lambda expressions evaluate to anonymous functions.

A Scheme list is written as elements in parentheses:

```
A number or a Pair with an operator as its first element
```

Lambda expressions evaluate to anonymous functions.

A Scheme list is written as elements in parentheses:

```
A number or a Pair with an operator as its first element
```

Lambda expressions evaluate to anonymous functions.

A Scheme list is written as elements in parentheses:

```
A number or a Pair with an operator as its first element
```

Lambda expressions evaluate to anonymous functions.

A Scheme list is written as elements in parentheses:

```
A number or a Pair with an operator as its first element
```

Lambda expressions evaluate to anonymous functions.