The interface for sets:
- Membership testing: Is a value an element of a set?
- Adjunction: Return a set with all elements in set1 and set2.
- 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>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td></td>
<td></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

<table>
<thead>
<tr>
<th>9</th>
<th>5</th>
<th>3</th>
</tr>
</thead>
</table>

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

Exceptions are raised with a raise statement.

`raise <expression>`

<Expression> must evaluate to an expression 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>:  
        <except 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 <except suite> is executed, with <name> bound to the exception.

A basic interpreter has two parts: a parser and an evaluator.

<table>
<thead>
<tr>
<th>string</th>
<th>parser</th>
<th>expression tree</th>
<th>Evaluator</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'add(2, 2)'</td>
<td>Exp('add', [2, 2])</td>
<td></td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>

An expression tree is a (hierarchical) data structure that represents a (nested) expression.

```python
class Exp(object):
    """A call expression in Calculator."""
    def __init__(self, operator, operands):
        self.operator = operator
        self.operands = operands

def calc_parse(line):
    """Parse a line of calculator input."""
    expression_tree = analyze_tokens(tokens=line)

def calc_apply(operator, arguments):
    """Apply the named operator to a list of args."""
    if operator == 'calc> ':
        return calc_eval(''.join(arguments))
    return calc_eval(operator + ''.join(arguments))

def calc_eval(exp):
    """Evaluate a Calculator expression."""
    if type(exp) in (int, float):
        return exp
    else:
        tokens = tokenize(exp)
        return calc_apply(tokens[0], tokens[1])
```

Lexical analyzer: Analyzes an input string as a sequence of tokens, which are symbols and delimiters.

Syntactic analyzer: Analyzes a sequence of tokens as an expression tree, which typically includes call expressions.

```python
tokenize('add(2, mul(4, 6))')

['add', ['(', '2', ')'], ['mul', ['(', '4', ')'], ['(', '6', ')']]]
```

```
Symbol: a built-in operator name  
Delimiter  
Symbol: a literal  
Delimiter
```

```
Properties
- numbers
- delimiters
- symbols
```

```
Builtin operators:
- +
- *
- <=
- %
```

```
Expression tokens:
- Numbers
- Operators
- Delimiters
```

```
Syntax tree:
- Numbers
- Operators
- Delimiters
```

```python
class LetterIterable(object):
    """A generator function."""
    def __iter__(self):
        current = 'a'
        while current <= 'd':
            yield current
            current = chr(ord(current) + 1)

class Letters(object):
    """An iterator over letters."""
    def __init__(self, letters):
        self.current = 'a'
        self.letters = letters
    def __next__(self):
        if self.current > 'd':
            raise StopIteration
        result = self.current
        self.current = chr(ord(result) + 1)
        return result
    def __iter__(self):
        return self

def letters_generator():
    """A generator function."""
    current = 'a'
    while current <= 'd':
        yield current
        current = chr(ord(current) + 1)

class Override_printloop:
    """Run a read-eval-print loop for calculator."""
    def __init__(self):
        self.current = 'a'
        self.letters = letters
    def __next__(self):
        return self.current
        self.current = chr(ord(self.current) + 1)
    def __iter__(self):
        return self
```

```
if item in Letters():
    print(item)
```

```
if not item in Letters():
    print('Item not in letters')
```

```python
class FilterStream(object):
    def __init__(self, fn, s):
        self.fn = fn
        self.s = s
    def __iter__(self):
        return self.filter_stream(self.fn, self.s)

def filter_stream(fn, s):
    return fn(s).

def compute_rest(first):
    return first._rest

def _init_(self, operator, operands):
    self.operator = operator
    self.operands = operands

def calc_parse(line):
    expression_tree = analyze_tokens(tokens=line)

def calc_apply(operator, arguments):
    if operator == 'calc> ':
        return calc_eval(''.join(arguments))
    return calc_eval(operator + ''.join(arguments))

def calc_eval(exp):
    if type(exp) in (int, float):
        return exp
    else:
        tokens = tokenize(exp)
        return calc_apply(tokens[0], tokens[1])

def calc_apply(operator, arguments):
    if operator == 'calc> ':
        return calc_eval(''.join(arguments))
    return calc_eval(operator + ''.join(arguments))

def calc_eval(exp):
    if type(exp) in (int, float):
        return exp
    else:
        tokens = tokenize(exp)
        return calc_apply(tokens[0], tokens[1])
```

```
Symbol: a built-in operator name  
Delimiter  
Symbol: a literal  
Delimiter
```

```
Builtin operators:
- +
- *
- <=
- %
```

```
Expression tokens:
- Numbers
- Operators
- Delimiters
```

```
Syntax tree:
- Numbers
- Operators
- Delimiters
```

```
Properties
- numbers
- delimiters
- symbols
```

```
Syntax tree:
- Numbers
- Operators
- Delimiters
```

```python
class LetterIterable(object):
    """A generator function."""
    def __iter__(self):
        current = 'a'
        while current <= 'd':
            yield current
            current = chr(ord(current) + 1)

class Letters(object):
    """An iterator over letters."""
    def __init__(self, letters):
        self.current = 'a'
        self.letters = letters
    def __next__(self):
        if self.current > 'd':
            raise StopIteration
        result = self.current
        self.current = chr(ord(result) + 1)
        return result
    def __iter__(self):
        return self

def letters_generator():
    """A generator function."""
    current = 'a'
    while current <= 'd':
        yield current
        current = chr(ord(current) + 1)
```

```
if item in Letters():
    print(item)
```

```
if not item in Letters():
    print('Item not in letters')
```
The run procedure evaluates a sentence as a line of Logo code and outputs its value:

```
? run [print sum 1 2]
3
? print run sentence "sum" sentence 10 [run (difference 7 3)];
14
```

Procedure definition is a special form, not a call expression:
```
Procedure name  Formal parameter
Body
```

Values bound to names are looked up using variable expressions:
```
? to double: x
? print double 4
8
```

Logo binds variable names to values, as does Python. An environment stores name bindings in a sequence of frames. Each frame can have at most one value bound to a given name. The make procedure adds or changes variable bindings:
```
? make "x 2
```

```
Values bound to names are looked up using variable expressions:
```
? print :x
2
```

```
Values bound to names are looked up using variable expressions:
```
? print :x
2
```

```
Values bound to names are looked up using variable expressions:
```
? print :x
2
```

```
Values bound to names are looked up using variable expressions:
```
? print :x
2
```

```
Values bound to names are looked up using variable expressions:
```
? print :x
2
```

```
Values bound to names are looked up using variable expressions:
```
? print :x
2
```

```
Values bound to names are looked up using variable expressions:
```
? print :x
2
```

```
Values bound to names are looked up using variable expressions:
```
? print :x
2
```

```
Values bound to names are looked up using variable expressions:
```
? print :x
2
```