INSTRUCTIONS

- You have 3 hours to complete the exam.
- The exam is closed book, closed notes, closed computer, closed calculator, except one hand-written 8.5” × 11” crib sheet of your own creation and the three official 61A study guides attached to the back of this exam.
- Mark your answers ON THE EXAM ITSELF. If you are not sure of your answer you may wish to provide a brief explanation.

Last name

First name

SID

Login

TA & section time

Name of the person to your left

Name of the person to your right

All the work on this exam is my own. (please sign)

For staff use only

<table>
<thead>
<tr>
<th>Q. 1</th>
<th>Q. 2</th>
<th>Q. 3</th>
<th>Q. 4</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>/20</td>
<td>/16</td>
<td>/30</td>
<td>/14</td>
<td>/80</td>
</tr>
</tbody>
</table>
1. (20 points) Bank Rewrite

(a) (10 pt) For each of the following expressions, write the `repr` string of the value to which the expression evaluates. Special cases: If an expression evaluates to a function, write FUNCTION. If evaluation would never complete, write FOREVER. If an error would occur, write ERROR.

Assume that the expressions are evaluated in order. Evaluating the first may affect the value of the second, etc.

Assume that you have started Python 3 and executed the following statements:

```
jan = [1, 3, 5]
feb = [3, 5, 7]

def mar(apr, may):
    if not apr or not may:
        return []
    if apr[0] == may[0]:
        return mar(apr[1:], may[1:]) + [apr[0]]
    elif apr[0] < may[0]:
        return mar(apr[1:], may)
    else:
        return mar(apr, may[1:])
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Evaluates to</th>
</tr>
</thead>
<tbody>
<tr>
<td>5*5</td>
<td>25</td>
</tr>
<tr>
<td><code>feb[jan[0]]</code></td>
<td></td>
</tr>
<tr>
<td><code>mar(jan, feb)</code></td>
<td></td>
</tr>
<tr>
<td><code>jan</code></td>
<td></td>
</tr>
<tr>
<td><code>next(iter(jan))</code></td>
<td></td>
</tr>
<tr>
<td><code>len(mar(range(5, 50), range(20, 200)))</code></td>
<td></td>
</tr>
</tbody>
</table>
(b) (10 pt) For each of the following expressions, write the `repr` string of the value to which the expression evaluates. Special cases: If an expression evaluates to a function, write FUNCTION. If evaluation would never complete, write FOREVER. If an error would occur, write ERROR.
Assume that you have started Python 3 and executed the following statements after executing the Stream class statement from the Final Exam Study Guide:

```python
from operator import add, mul

def stone(a):
    return Stream(a, lambda: stone(a+1))
rock = stone(3)

def lava(x, y, z):
    def magma():
        return lava(x.rest, y.rest, z)
    volcano = z(x.first, y.first)
    return Stream(volcano, magma)
fire = lava(rock, rock.rest, mul)

def hot():
    crater = Stream(0, lambda: lava(crater, rock, add))
    return crater
ash = hot()
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Evaluates to</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, rock.first)</td>
<td>(1, 3)</td>
</tr>
<tr>
<td>(rock.rest.first, rock.rest.rest.first)</td>
<td></td>
</tr>
<tr>
<td>(fire.first, fire.rest.first)</td>
<td></td>
</tr>
<tr>
<td>fire.rest is fire.rest</td>
<td></td>
</tr>
<tr>
<td>(ash.first, ash.rest.first)</td>
<td></td>
</tr>
<tr>
<td>ash.rest.rest.rest.first</td>
<td></td>
</tr>
</tbody>
</table>
2. (16 points) Web Rater Ink

(a) (8 pt) Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. You may not need to use all of the spaces or frames.

A complete answer will:
- Add all missing names, labels, and parent annotations to all local frames.
- Add all missing values created during execution.
- Show the return value for each local frame.

```python
def snow(x):
    def ice(x):
        if x == 0:
            return 1
        return 2 + rain(ice, x)
    def rain(g, h):
        return 3 + g(h - x)
    return ice(x)
snow(4)
```

Return Value

Return Value

Return Value

Return Value
(b) (8 pt) Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. You may not need to use all of the spaces or frames.

A complete answer will:

- Add all missing names, labels, and parent annotations to all local frames.
- Add all missing values created during execution.
- Show the return value for each local frame.

```python
def box(a):
    def box(b):
        def box(c):
            nonlocal a
            a = a + c
            return (a, b)
        return box
    gift = box(1)
    return (gift(2), gift(3))

box(4)
```

Return Value

Return Value

Return Value

Return Value
3. (30 points) Twin Breaker

(a) (8 pt) Run-length encoding (RLE) is a technique used to compress sequences that contain repeated elements. For example, the sequence 1,1,4,2,2,2 would be encoded as three 1's, one 4, and four 2's. Fill in the blanks in the RLE class below, so that all doctests pass.

class RLE(object):
    """A run-length encoding of a sequence."

    >>> RLE([2, 2, 2, 2, 2, 2, 7]).runs
    [(5, 2), (1, 7)]
    >>> s = RLE([1, 1, 1, 4, 2, 2, 2, 2])
    >>> s.runs
    [(3, 1), (1, 4), (4, 2)]
    >>> len(s)
    8
    >>> s[2], s[3], s[4], s[5]
    (1, 4, 2, 2)
    ""

    def __init__(self, elements):
        last, count = None, 0
        self.runs = []

        for elem in elements:
            if ________________________________:
                self.runs.append(__________________)  # Fill this blank

            if ________________________________:
                last, count = _______________________

            else:
                count += 1

    def __len__(self):
        return sum(________________________)

def __getitem__(self, k):
    run = 0

    while ____________________________:
        k, run = _________________________

    return self.runs[run][1]
(b) (6 pt) A path through a tree is a sequence of connected nodes in which each node appears at most once. The height of a tree is the longest path starting at the root. Fill in the blanks in the calls to max below, so that all doctests pass. Write each operand expression on a separate line. **You may not need to use all of the blank lines.** This question uses the `Tree` class statement from the Midterm 2 Study Guide. You may assume that `height` works correctly when implementing `longest`.

```python
s = Tree(0, Tree(1, Tree(2, Tree(3), Tree(4))))
t = Tree(5, Tree(6, Tree(7, s, Tree(8)), Tree(9, None, Tree(10, s))))
```

def height(tree):
    """Return the length of the longest path from the root to a leaf."

    >>> height(None)
    0
    >>> height(s)
    4
    >>> height(t)
    8
    """

    if tree is None:
        return 0
    return 1 + max(___________________________________________________
                    _____________________________________________
                    _____________________________________________)

def longest(tree):
    """Return the length of the longest path between any two nodes."

    >>> longest(None)
    0
    >>> longest(Tree(5))
    1
    >>> [longest(b) for b in (s.left.left, s.left, s)]
    [3, 3, 4]
    >>> longest(t)
    12
    """

    if tree is None:
        return 0
    return max(___________________________________________________
                _____________________________________________
                _____________________________________________
                _____________________________________________)
(c) (8 pt) Given a set of unique positive integers \( s \) and a maximum sum \( m \), the `pack` function returns a subset of \( s \) with the largest sum less than or equal to \( m \). Fill in the blanks below, so that all doctests pass.

Assume that sets are printed in sorted order, regardless of how they are constructed.

```python
def pack(s, m):
    """Return the subset of \( s \) with the largest sum up to \( m \)."

    >>> s = [4, 1, 3, 5]
    >>> pack(s, 7)
    {3, 4}
    >>> pack(s, 6)
    {1, 5}
    >>> pack(s, 11)
    {1, 4, 5}
    ""

    if len(s) == 0:
        return set()
    if s[0] > m:
        return ________________________________
    with_s0 = {s[0]}.union(__________________________)
    without = _________________________________
    if _________________________________:
        return with_s0
    else:
        return without
```
(d) (8 pt) Cross out lines from the implementation of the \texttt{IterableTree} class below so that all doctests pass and the implementation contains as few lines of code as possible. Don’t cross out any docstrings or doctests.

The \texttt{__iter__} generator for this class should yield the entries of the tree (and each subtree) starting with the root, and yield all of the entries of the left branch before any of the entries of the right branch. This question uses the \texttt{Tree} class statement from the Midterm 2 Study Guide.

```python
class IterableTree(object):
    def __init__(self, entry, left=None, right=None):
        Tree.__init__(entry, left, right)
        Tree.__init__(self, entry, left, right)
    def __iter__(self):
        """Yield the entries of this tree."
        yield self.entry
        yield entry
        for branch in (self.left, self.right):
            if branch:
                if self.branch:
                    branch = iter(branch)
                    for entry in branch:
                        for entry in branch():
                            yield self.entry
                            yield entry
                            yield self.entry
```

```python
>>> T = IterableTree
>>> t = T('A', T(2, T('C'), T(4)), T('E', None, T(6)))
>>> list(t)
['A', 2, 'C', 4, 'E', 6]
"""
```
4. (14 points) Winter Break

(a) (2 pt) Write the value of the Scheme expression (f 7) after evaluating the define expressions below?

```
(define j
  (mu (c k)
    (if (< c n)
      (j (+ c 2) (- k 1))
      k)))

(define (f n)
  (define c n)
  (j 0 0))
```

(b) (2 pt) Circle (True or False): Every call to j above is a tail call.

(c) (4 pt) In your project 4 implementation, how many total calls to scheme_eval and scheme_apply would result from evaluating the following two expressions? Assume that you are not using the tail call optimized scheme_eval optimized function for evaluation.

```
(define (square x) (* x x))
(+ (square 3) (- 3 2))
```

Calls to scheme_eval (circle one): 2 5 14 24

Calls to scheme_apply (circle one): 1 2 3 4
(d) (4 pt) Fill in two facts below to complete the definitions of the relations reversed and palindrome. The reversed relation indicates that the first list contains the same elements as the second, but in reversed order. The palindrome relation indicates that a list is the same backward and forward.

logic> (fact (append-to-form () ?x ?x))

logic> (fact (append-to-form (?a . ?r) ?y (?a . ?z))
        (append-to-form ?r ?y ?z))

logic> (fact (reversed () ()))

logic> (fact (reversed (?a . ?r) ?s)
        (reversed ?r ?rev))

logic> (query (reversed ?x (a b c d)))
Success!
x: (d c b a)

logic> (fact (palindrome ?s))

logic> (query (palindrome (a b ?x d e ?y ?z)))
Success!
x: e   y: b   z: a

(e) (2 pt) Define a simple mathematical function \( f(n) \) such that calling \( m(n) \) on positive integer \( n \) prints \( \Theta(f(n)) \) lines of output.

```python
def m(n):
    g(n)
    if n <= 2:
        print('The')
    else:
        m(n//3)

def g(n):
    if n == 42:
        print('Last')
    if n <= 0:
        print('Question')
    else:
        g(n-1)
```

\( f(n) = \)
Execution rule for while statements:

1. Evaluate the header's expression.
2. If the result is a true value, execute the suite, then return to step 1.
3. Otherwise, evaluate the <left> expression, then skip the suite.

Evaluation rule for not expressions:

1. Evaluate the subexpression <left>.
2. If the result is a true value v, then the expression evaluates to the value of the operand subexpressions.
3. Otherwise, the expression evaluates to the value of the operand subexpressions.

Evaluation rule for and expressions:

1. Evaluate the subexpression <left>.
2. If the result is a true value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the operand subexpressions.

Evaluation rule for or expressions:

1. Evaluate the subexpression <left>.
2. If the result is a true value v, then the expression evaluates to v.
3. Otherwise, evaluate the subexpression <right>.
4. If the result is a true value v, then the expression evaluates to v.
5. Otherwise, the expression evaluates to the value of the operand subexpressions.

Execution rule for conditional statements:

1. Evaluate the header's expression.
2. If it is a true value, execute the suite, then return to step 1.
3. Otherwise, evaluate the remaining clauses in the statement.

Execution rule for def statements:

1. Create a new function value with the specified name, formal parameters, and function body.
2. Its parent is the first frame of the current environment.
3. Bind the name of the function to the function value in the first frame of the current environment.

Execution rule for assignment statements:

1. Evaluate the expression(s) on the right of the equal sign.
2. Simultaneously bind the names on the left to those values, in the frame of the current environment.

Execution rule for conditional statements:

1. Each clause is considered in order.
2. If it is a true value, execute the suite, then skip the remaining clauses in the statement.

Execution rule for for expressions:

1. Evaluate the subexpression <left>.
2. If the result is a false value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the operand subexpressions <right>.

Evaluation rule for while expressions:

1. Evaluate the subexpression <left>.
2. If the result is a false value v, then the expression evaluates to v.
3. Otherwise, evaluate the subexpression <right>.

Evaluation rule for not expressions:

1. Evaluate exp; The value is True if the result is a false value, and False otherwise.

Execution rule for while statements:

1. Evaluate the header's expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.
3. Otherwise, evaluate the remaining clauses in the statement.

Evaluation rule for call expressions:

1. Call the function that is the value of the operator.
2. Apply the function to the arguments.
3. Evaluate the operator and operand subexpressions.

Evaluation rule for call expressions:

1. Evaluate the operator and operand subexpressions.
2. Apply the function that is the value of the operator subexpression to the arguments that are the values of the operand subexpressions.

Applying user-defined functions:

1. Create a new local frame with the same parent as the function that was applied.
2. Bind the arguments to the function's formal parameter names in that frame.
3. Execute the body of the function in the environment beginning at that frame.

Evaluation rule for call expressions:

1. Evaluate the header's expression.
2. If it is a true value, execute the suite, then skip the remaining clauses in the statement.
3. Otherwise, evaluate the remaining clauses in the statement.

Evaluation rule for and expressions:

1. Evaluate the subexpression <left>.
2. If the result is a false value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the operand subexpressions <right>.

Evaluation rule for or expressions:

1. Evaluate the subexpression <left>.
2. If the result is a false value v, then the expression evaluates to v.
3. Otherwise, evaluate the subexpression <right>.
4. If the result is a true value v, then the expression evaluates to v.
5. Otherwise, the expression evaluates to the value of the operand subexpressions <right>.

Evaluation rule for not expressions:

1. Evaluate exp; The value is True if the result is a false value, and False otherwise.
A function with formal parameters x and y and body "return x * y"
Must be a single expression

```python
def make_adder(k):
    def adder(k):
        return k + n
    return adder

>>> add_three = make_adder(3)
>>> add_three(4)
7
```

A function that returns a function

```python
def square(x):
    return x * x

def adder(k):
    return k + n
```

The name add_three is bound to a function

```python
def triple(x):
    return x * 3

triple = make_adder(3)
```

A local def statement
Can refer to names in the enclosing function

```python
def triple(x):
   return x * 3

# Call triple
triple(5)
```

• Every user-defined function has a parent frame
• The parent of a function is the frame in which it was defined
• Every local frame has a parent frame
• The parent of a frame is the parent of the function called

- Compound objects combine objects together
- An abstract data type lets us manipulate compound objects as units
- Programs that use data isolate two aspects of programming:
  - How data are represented (as parts)
  - How data are manipulated (as units)

- Data abstraction: A methodology by which functions enforce an abstraction barrier between representation and use

```python
def square(x):
    return x * x
```

What does sum_squares need to know about square?
- Square takes one argument. Yes
- Square has the intrinsic name square. No
- Square computes the square of a number. Yes
- Square computes the square by calling square. No

```python
def sum_squares(x, y):
    return square(x) + square(y)
def square(x):
    return x * x
```

Begin with a function f and an initial guess x
1. Compute the value of f at the guess: f(x)
2. Compute the derivative of f at the guess: f'(x)
3. Update guess to be: x' = f(x) - f'(x)

```python
def approx_derivative(f, x):
    return f(x)

def newton_update(f):
    def update(x):
        k = 0
        while not done(guess) and k < max_updates:
            guess = update(guess)
            k += 1
        return guess

def divide_exact(N, D):
    q, r = divide_exact(N, D)
    return q, r

>>> q, r = divide_exact(2012, 10)
>>> q
201
>>> r
2
```

How to find the square root of 2?
1. Begin with a function f and an initial guess x
   1. Compute the value of f at the guess: f(x) = x^2 - 2
   2. Compute the derivative of f at the guess: f'(x) = 2x
   3. Update guess to be: x' = f(x) - f'(x) = x^2 - 2 - 2x

```python
def find_root(f, guess, max_updates):
    def done(guess):
        return f(guess) < 0.001
    return iter_improve(newton_update(f), guess)

>>> find_root(square, 1.0, 100)
1.4142135623730951
```

Iteratively improve guess with update until done returns a true value.

```python
def iter_improve(update, done, guess, max_updates):
    def max_updates(guess):
        return done(guess)

    def return_update(update, done, guess):
        def update(guess):
            k = 0
            while not done(guess) and k < max_updates:
                guess = update(guess)
                k += 1
            return guess

    return_update(update, done, guess)

>>> iter_improve(square, guess, 1.0, 100)
1.4142135623730951
```

A function's signature has all the information to create a local frame
• Tuples are immutable sequences.
• Lists are mutable sequences.
• Dictionaries are unordered collections of key-value pairs.

Dictionary keys do have two restrictions:
• A key of a dictionary cannot be an object of a mutable built-in type.
• Two keys cannot be equal. There can be at most one value for a key.

Dictionaries are unordered:
- Two of key-value pairs.
- Size of the problem evaluated when the list comprehension is evaluated.

Unlike generator expressions, the map expression is evaluated when the generator expression is evaluated.

Elements selection
- A sequence has a finite length.
- A sequence has an element corresponding to any non-negative integer less than its length, starting at 0 for the first element.

Generator expressions
- Evaluates to an iterable object.
- if <filter exp> is evaluated when the generator expression is evaluated.
- Remaining expressions are evaluated when elements are accessed.

List comprehensions
- [map exp for <name> in <iter exp> if <filter exp>]
- Short version: <map exp for <name> in <iter exp>]

Unlike generator expressions, the map expression is evaluated when the list comprehension is evaluated.

Mutable values can be changed without a nonlocal statement:
- The def statement header is similar to other functions.
- Identity testing is performed by "is" and "is not" operators.
- Conditional statements check for base cases.
- Base cases are evaluated without recursive calls.
- Typically, all other cases are evaluated with recursive calls.
To evaluate a dot expression: <expression> . <name>
1. Evaluate the expression to the left of the dot, which yields the object of the dot expression.
2. <name> is matched against the instance attributes of that object; if an attribute with that name exists, its value is returned.
3. If not, <name> is looked up in the class, which yields a class attribute value.
4. That value is returned unless it is a function, in which case a bound method is returned instead.

To look up a name in a class:
1. If it names an attribute in the class, return the attribute value.
2. Otherwise, look up the name in the base class, if there is one.

Assignment statements with a dot expression on the left-hand side affect the object for the object of that dot expression:
- If the object is an instance, then assignment sets an instance attribute.
- If the object is a class, then assignment sets a class attribute.

When a class is called:
1. A new instance of that class is created.
2. The constructor __init__ of the class is called with the new object as its first argument (called self), along with additional arguments provided in the call expression.

```python
class Account(object):
    interest = 0.02

    def __init__(self, account_holder):
        self.balance = 0
        self.account_holder = account_holder

    def deposit(self, amount):
        self.balance += amount
        return self.balance

    def withdraw(self, amount):
        if amount <= self.balance:
            return 'Insufficient funds'
        self.balance -= amount
        return self.balance

>>> jim_account = Account('Jim')  # Instantiated Account class
>>> jim_account.interest  # Method on account instance
0.02

>>> tom_account = Account('Tom')  # Instantiated Account class
>>> tom_account.interest  # Method on account instance
0.02

```

```
class CheckingAccount(Account):
    withdraw_fee = 1

    def withdraw(self, amount):
        return Account.withdraw(self, amount + self.withdraw_fee)

```

```
class SavingsAccount(Account):
    deposit_fee = 2

    def deposit(self, amount):
        return Account.deposit(self, amount - self.deposit_fee)

```

```
class AsSeenOnTvAccount(Account, SavingsAccount):
    def __init__(self, account_holder):
        self.balance = account_holder

```

```
class Rlist(object):
    class EmptyList(object):
        def __len__(self):
            return 0

        empty = EmptyList()

    def __len__(self):
        return 1 + len(self.rest)

    def getitem(self, i):
        if i == 0:
            return self.first
        return self.rest[i-1]

```

```
class Tree(object):
    def __init__(self, kind, left=None, right=None):
        self.kind = kind
        self.left = left
        self.right = right

    def map_rlist(s, fn):
        if s is Rlist:
            return Rlist(map_rlist(s.left, fn), map_rlist(s.right, fn))

    def count_leaves(self):
        if type(self) is list:
            return 1 + sum(map(count_leaves, tree))
```
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>Intersection</th>
<th>Union</th>
<th>Adjunction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</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
- Smaller than all entries in its right branch

```
if 9 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.

```
try:
    <expression>
except <exception class> as <name>:
    <expression>
...
```

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.

```
class Stream(object):
    """A lazily computed recursive list."""
    def __new__(self):
        return Stream.empty
    def __init__(self, first, compute_rest=lambda: Stream.empty):
        self.first = first
        self.compute_rest = compute_rest
    def __repr__(self):
        return "Stream({!r}, compute_rest={!r})".format(self.first, self.compute_rest)
    @property
    def rest(self):
        return self._compute_rest() if self._compute_rest is not None else self
    def __next__(self):
        return self.first
    def __iter__(self):
        return Stream.empty
    def __next__(self):
        return self.first
    def __iter__(self):
        return Stream(self.first, compute_rest=self.compute_rest)
```

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
    log0: (fact (parent delano herbert))
    log1: (fact (parent abraham barack))
    log2: (fact (parent abraham clinton))
    log3: (fact (parent fillmore delano))
    log4: (fact (parent fillmore grover))
    log5: (fact (parent eisenhower fillmore))
    ```

  Relations can contain relations in addition to atoms.
    ```python
    log0: (fact (dog (name abraham) (color white)))
    log1: (fact (dog (name barack) (color tan)))
    log2: (fact (dog (name clinton) (color white)))
    log3: (fact (dog (name delano) (color white)))
    log4: (fact (dog (name eisenhower) (color tan)))
    log5: (fact (dog (name fillmore) (color brown)))
    ```

  Variables can refer to atoms or relations in queries.
    ```python
    log0: (query (parent abraham ?child))
    Success!
    child: barack
    ```

  Fail.
    ```python
    log0: (query (child ?fillmore))
    Success!
    child: fillmore
    ```

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

**Two lists append to form a third list:

- 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
  log0: (fact (append-to (1) (7) (x)))
  log1: (fact (append-to (7a) (7r) (y) (7a) (7z)))
  ```
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>)

An iterator object has a method __iter__ that returns an iterator.

```python
>>> (def pi 3.14)
>>> (def (abs x) (- x))
>>> (def (length-iter (x y z)) (+ x y (square z)))
>>> (define (abs x) (- x))
>>> (define (square x) (* x x))
```

Lexical scope:
The way in which names are looked up in Scheme and Python is...

- Dots can be used in a quoted list to specify the second...
- • They also used a non-obvious notation for recursive lists.
- • In the late 1950s, computer scientists used confusing names.
- (define plus4 (lambda (x) (+ x 4)))
- Two equivalent expressions:
  
  - (lambda (x y z) (+ x y (square z)))
  - (lambda (x y z) (+ x y (square z)))

Lambda expressions evaluate to anonymous functions.

```python
(lambda (x y z) (+ x y (square z)))
```

Two equivalent expressions:

- (define plus4 (lambda (x) (+ x 4)))
- (define plus4 (lambda (x) (+ x 4)))

An operator can be a call expression too:

```
(lambda (y z) (+ y z))
```

In the late 1950s, computer scientists used confusing names.

- cons: Two-argument procedure that creates a pair.
- car: Procedure that returns the first element of a pair.
- cdr: Procedure that returns the second element of a pair.
- nil: The empty list

They also used a non-obvious notation for recursive lists.

- A (recursive) Scheme list is a pair in which the second element is nil or a Scheme list.
- Scheme lists are written as space-separated combinations.
- A dotted list has an arbitrary value for the second element of the last pair. Dotted lists may not be well-formed lists.

```
> (define x (cons 1 2))
> x
(1 2)
> (car x)
1
> (cdr x)
2
> (cons 1 (cons 2 (cons 3 nil)))
(1 2 3)
```

Symbols normally refer to values; how do we refer to symbols?

```
> (define a 1)
> (define b 2)
> (list a b)
(1 2)
```

Quotation is used to refer to symbols directly in Lisp.

```
> (list 'a 'b)
(l 'a 'b)
```

Quotation can also be applied to combinations to form lists.

```
> (car (a b c))
(a b c)
> (cdr (a b c))
(b c)
```

Dots can be used in a quoted list to specify the second element of the final pair.

```
> (cdr (cdr '(12 3)))
3
```

However, dots appear in the output only of ill-formed lists.

```
> '(12 . 3)
(1 2 3)
> '(12 . (3 4))
(1 2 3 4)
> '(12 . nil)
(1 2)
> (cdr '(12 . (3 4 . (5))))
(3 4 5)
```

The way in which names are looked up in Scheme and Python is called lexical scope (or static scope).

**Lexical scope:** The parent of a frame is the environment in which a procedure was defined. (lambda ...)

**Dynamic scope:** The parent of a frame is the environment in which a procedure was called. (mu ...)

```
> (define f (mu (x) (+ x y)))
> (define g (lambda (y) (f (+ x y))))
> (g 3 7)
13
```

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

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

```
Lines Parser expression Evaluator value

('(+ 2 3)') Pair('(', Pair(2, Pair(3, nil)))) 4
('(+ 3 1)' Pair('(', Pair(3, Pair(1, ...))...)) 4
('(- 23)' Pair('(', Pair('-', Pair(23, ...))...)) 18

Lines forming a Scheme expression
A number or a Pair with an operator as its first element
A number

A Scheme list is written as elements in parentheses:

```
(define x (cons 1 2))
(define y (cons 3 4))
(define z (cons 5 6))
```

Each element can be a combination or atom (primitive).

• (+ 3 (+ 2 4) (+ 3 5)) (+ 10 7) 6)

The task of parsing a language involves coercing a string representation of an expression to the expression itself.

Parsers must validate that expressions are well-formed.

A Parser takes a sequence of lines and returns an expression.

```
Define (primitive values) (numbers)
Look up values bound to symbols
Recursive calls:
Evaluate operands of call expressions
Apply (operator | arguments)
Evaluate (sub-expressions) of special forms
```

```
Lexical analysis Tokens Syntactic analysis Expression

('(', '*', 4, 5.6, ')', ')', '(', '-', 23, ')', '(', '+', 1, ...))

Expression Trees
```

The structure of the Scheme interpreter

```
Apply

Base cases:
- Built-in primitive procedures
- Eval(body) of user-defined proc’s
```

```
Eval
```

```
Requires an environment for name lookup
```

```
LambdaProcedure instance [parent=g]
```

```
A procedure call that has not yet returned is active. Some
procedure calls are tail calls. A Scheme interpreter should
support an unbounded number of active tail calls.
```

```
A tail call is a call expression in a tail context, which are:
- The last body expression in a lambda expression
- Expressions 2 & 3 (consequent & alternative) in a tail context
```

```
if expression
```

```
(define (factorial n k)
  (if (zero? n) k
      (factorial (- n 1) (+ k n))))
```

```
(define (length-tail s)
  (define (length-iter s n)
    (if (null? s) n
        (+ 1 (length-iter (cdr s) (+ n 1))))
    (length-iter s 0))
```

To apply a user-defined procedure, create a new frame in which
formal parameters are bound to argument values, whose parent
is the env of the procedure, then evaluate the body of the
procedure in the environment that starts with this new frame.

```
(f (list 1 2))
```

```
(g: Global frame

([parent=g]

([parent=g] + - 0 = nil

([parent=g] + - 0 = nil

A procedure call that has not yet returned is active. Some
procedure calls are tail calls. A Scheme interpreter should
support an unbounded number of active tail calls.
```

```
A tail call is a call expression in a tail context, which are:
- The last body expression in a lambda expression
- Expressions 2 & 3 (consequent & alternative) in a tail context
```

```
if expression
```

```
(define (factorial n)
  (if (zero? n) 1
      (* n (factorial (- n 1)))))
```

```
(define (length-tail s)
  (define (length-iter s n)
    (if (null? s) n
        (+ 1 (length-iter (cdr s) (+ n 1))))
    (length-iter s 0))
```

A recursive call is a tail call

```
```