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 3 official 61A midterm 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.

<table>
<thead>
<tr>
<th>Last name</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>First name</td>
<td></td>
</tr>
<tr>
<td>SID</td>
<td></td>
</tr>
<tr>
<td>Login</td>
<td></td>
</tr>
<tr>
<td>TA &amp; section time</td>
<td></td>
</tr>
<tr>
<td>Name of the person to your left</td>
<td></td>
</tr>
<tr>
<td>Name of the person to your right</td>
<td></td>
</tr>
<tr>
<td>All the work on this exam is my own. (please sign)</td>
<td></td>
</tr>
</tbody>
</table>

For staff use only

<table>
<thead>
<tr>
<th>Q. 1</th>
<th>Q. 2</th>
<th>Q. 3</th>
<th>Q. 4</th>
<th>Q. 5</th>
<th>Q. 6</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>/14</td>
<td>/16</td>
<td>/12</td>
<td>/12</td>
<td>/18</td>
<td>/8</td>
<td>/80</td>
</tr>
</tbody>
</table>
1. (14 points) Representing Scheme Lists

For each row below, write the output displayed by the interactive Python interpreter when the expression is evaluated. Expressions are evaluated in order, and expressions may affect later expressions.

Whenever the interpreter would report an error, write Error. You should include any lines displayed before an error. Reminder: The interactive interpreter displays the repr string of the value of a successfully evaluated expression, unless it is None.

The Pair class from Project 4 is described on your final study guide. Recall that its __str__ method returns a Scheme expression, and its __repr__ method returns a Python expression. The full implementation of Pair and nil appear at the end of the exam as an appendix. Assume that you have started Python 3, loaded Pair and nil from scheme_reader.py, then executed the following:

```
blue = Pair(3, Pair(4, nil))
gold = Pair(Pair(6, 7), Pair(8, 9))
```

def process(s):
    cal = s
    while isinstance(cal, Pair):
        cal.bear = s
        cal = cal.second
    if cal is s:
        return cal
    else:
        return Pair(cal, Pair(s.first, process(s.second)))

def display(f, s):
    if isinstance(s, Pair):
        print(s.first, f(f, s.second))

y = lambda f: lambda x: f(f, x)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pair(1, nil)</td>
<td>Pair(1, nil)</td>
</tr>
<tr>
<td>print(Pair(1, nil))</td>
<td>(1)</td>
</tr>
<tr>
<td>1/0</td>
<td>Error</td>
</tr>
<tr>
<td>print(print(3), 1/0)</td>
<td></td>
</tr>
<tr>
<td>print(Pair(2, blue))</td>
<td></td>
</tr>
<tr>
<td>print(gold)</td>
<td></td>
</tr>
<tr>
<td>process(blue.second)</td>
<td></td>
</tr>
<tr>
<td>print(process(gold))</td>
<td></td>
</tr>
<tr>
<td>gold.second.bear.first</td>
<td></td>
</tr>
<tr>
<td>y(display)(gold)</td>
<td></td>
</tr>
</tbody>
</table>
2. (16 points) Environments

(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 and parent annotations to all local frames.
• Add all missing values created during execution.
• Show the return value for each local frame.

```python
def tattoo(heart):
    def mom():
        nonlocal mom
        mom = lambda: heart(2) + 1
        return 3
    return mom() + mom() + 4

tattoo(lambda ink: ink + 0.5)
```

---

![Environment Diagram](image-url)
(b) (6 pt) For the six-line program below, fill in the three environment diagrams that would result after executing each pair of lines in order. **You must use box-and-pointer diagrams to represent list values. You do not need to write the word “list” or write index numbers.**

**Important:** All six lines of code are executed in order! Line 3 is executed after line 2 and line 5 after line 4.

---

1. ```python
   meow = [1, 2]
   cat = [meow, [4, 5]]
```

2. ```python
   cat[0][1] = cat[1][0]
   cat[meow[0]][0] = meow
```

3. ```python
   meow[0] = [cat.pop(0)]
   cat.extend(cat[0][1:])
```

---

(c) (2 pt) Circle the value, **True** or **False**, of each expression below when evaluated in the environment created by executing all six lines above. If you leave this question blank, you will receive 1 point.

Circle **True** or **False**: meow is cat[0]

Circle **True** or **False**: meow[0][0] is cat[0][0]
3. (12 points) Expression Trees

Your partner has created an interpreter for a language that can add or multiply positive integers. Expressions are represented as instances of the `Tree` class and must have one of the following three forms:

- **(Primitive)** A positive integer entry and no branches, representing an integer
- **(Combination)** The entry '+', representing the sum of the values of its branches
- **(Combination)** The entry '*', representing the product of the values of its branches

The `Tree` class is on the Midterm 2 Study Guide. The sum of no values is 0. The product of no values is 1.

(a) (6 pt) Unfortunately, multiplication in Python is broken on your computer. Implement `eval_with_add`, which evaluates an expression without using multiplication. You may fill the blanks with names or call expressions, but the only way you are allowed to combine two numbers is using addition.

```python
def eval_with_add(t):
    """Evaluate an expression tree of * and + using only addition."

    >>> plus = Tree('+', [Tree(2), Tree(3)])
    >>> eval_with_add(plus)
    5
    >>> times = Tree('*', [Tree(2), Tree(3)])
    >>> eval_with_add(times)
    6
    >>> deep = Tree('*', [Tree(2), plus, times])
    >>> eval_with_add(deep)
    60
    >>> eval_with_add(Tree('*'))
    1
    """
    if t.entry == '+':
        return sum(________________________________________________________)
    elif t.entry == '*':
        total = __________________________________________________________
        for b in t.branches:
            total, term = 0, ______________________________________________
            for __________ in ____________________________________________:
                total = total + term
        return total
    else:
        return t.entry
```
(b) (6 pt) A TA suggests an alternative representation of an expression, in which the entry is the value of the expression. For combinations, the operator appears in the left-most (index 0) branch as a leaf.

Implement transform, which takes an expression and mutates all combinations so that their entries are values and their first branches are operators. In addition, transform should return the value of its argument. You may use the calc_apply function defined below.

```python
def calc_apply(operator, args):
    if operator == '+':
        return sum(args)
    elif operator == '*':
        return product(args)

def product(vals):
    total = 1
    for v in vals:
        total *= v
    return total

def transform(t):
    """Transform expression tree t to have value entries and operator leaves."

    >>> seven = Tree('+', [Tree('*'), [Tree(2), Tree(3)]], Tree(1))
    >>> transform(seven)
    7
    >>> seven
    Tree(7, [Tree('+'), Tree(6, [Tree('*'), Tree(2), Tree(3)]), Tree(1)])
    """

    if t.branches:
        args = []
        for b in t.branches:
            args.append(___________________________)

        t.branches = ____________________________

        t.entry = ______________________________

    return _________________________________
```
4. (12 points) Lazy Sunday

(a) (4 pt) A flat-map operation maps a function over a sequence and flattens the result. Implement the flat_map method of the FlatMapper class. You may use at most 3 lines of code, indented however you choose.

```python
class FlatMapper:
    
    """A FlatMapper takes a function fn that returns an iterable value. The flat_map method takes an iterable s and returns a generator over all values in the iterables returned by calling fn on each element of s.

    >>> stutter = lambda x: [x, x]
    >>> m = FlatMapper(stutter)
    >>> g = m.flat_map((2, 3, 4, 5))
    >>> type(g)
    <class 'generator'>
    >>> list(g)
    [2, 2, 3, 3, 4, 4, 5, 5]
    ""

    def __init__(self, fn):
        self.fn = fn

    def flat_map(self, s):
        for x in s:
            for y in self.fn(x):
                yield y
```

(b) (2 pt) Define cycle that returns a Stream repeating the digits 1, 3, 0, 2, and 4. Hint: \((3+2)\%5\) equals 0.

```python
def cycle(start=1):
    """Return a stream repeating 1, 3, 0, 2, 4 forever.

    >>> first_k(cycle(), 12) # Return the first 12 elements as a list
    [1, 3, 0, 2, 4, 1, 3, 0, 2, 4, 1, 3]
    ""

    def compute_rest():
        return  

    return Stream(___________________________, __________________________)
```
(c) (4 pt) Implement the Scheme procedure \texttt{directions}, which takes a number \texttt{n} and a symbol \texttt{sym} that is bound to a nested list of numbers. It returns a Scheme expression that evaluates to \texttt{n} by repeatedly applying \texttt{car} and \texttt{cdr} to the nested list. Assume that \texttt{n} appears exactly once in the nested list bound to \texttt{sym}.

\textit{Hint:} The implementation searches for the number \texttt{n} in the nested list \texttt{s} that is bound to \texttt{sym}. The returned expression is built during the search. See the tests at the bottom of the page for usage examples.

\begin{verbatim}
(define (directions n sym)
  (define (search s exp)
    ; Search an expression \texttt{s} for \texttt{n} and return an expression based on \texttt{exp}.
    (cond ((number? s) _________________________________________________________)
      ((null? s) nil)
      (else (search-list s exp))))
  (define (search-list s exp)
    ; Search a nested list \texttt{s} for \texttt{n} and return an expression based on \texttt{exp}.
    (let ((first __________________________________________________________)
      (rest __________________________________________________________))
      (if (null? first) rest first)))
  (search (eval sym) sym))

define a '(1 (2 3) ((4))))
(directions 1 'a)
; expect (car a)
(directions 2 'a)
; expect (car (car (cdr a)))
(define b '((3 4) 5))
(directions 4 'b)
; expect (car (cdr (car b)))

d(d) (2 pt) What expression will (directions 4 'a) evaluate to?
\end{verbatim}
5. (18 points) Basis Loaded

Ben Bitdiddle notices that any positive integer can be expressed as a sum of powers of 2. Some examples:

\[
\begin{align*}
11 &= 8 + 2 + 1 \\
23 &= 16 + 4 + 2 + 1 \\
24 &= 16 + 8 \\
45 &= 32 + 8 + 4 + 1 \\
2014 &= 1024 + 512 + 256 + 128 + 64 + 16 + 8 + 4 + 2
\end{align*}
\]

A basis is a linked list of decreasing integers (such as powers of 2) with the property that any positive integer \( n \) can be expressed as the sum of elements in the basis, starting with the largest element that is less than or equal to \( n \).

(a) (4 pt) Implement sum_to, which takes a positive integer \( n \) and a linked list of decreasing integers basis. It returns a linked list of elements of the basis that sum to \( n \), starting with the largest element of basis that is less than or equal to \( n \). If no such sum exists, raise an ArithmeticError. Each number in basis can only be used once (or not at all). The Link class is described on your Midterm 2 Study Guide.

```python
def sum_to(n, basis):
    """Return elements of linked list basis that sum to n."

    >>> twos = Link(32, Link(16, Link(8, Link(4, Link(2, Link(1))))))
    >>> sum_to(11, twos)
    Link(8, Link(2, Link(1)))
    >>> sum_to(23, twos)
    Link(16, Link(4, Link(2, Link(1))))
    >>> sum_to(24, twos)
    Link(16, Link(8))
    >>> sum_to(45, twos)
    Link(32, Link(8, Link(4, Link(1))))
    ""

    if ________________________________________________________________:
        return Link.empty

    elif ________________________________________________________________:
        raise ArithmeticError

    elif basis.first > n:
        return sum_to(n, basis.rest)

    else:
        return ________________________________________________________
```
(b) (6 pt) Cross out as many lines as possible in the implementation of the `FibLink` class so that all doctests pass. A `FibLink` is a subclass of `Link` that contains decreasing Fibonacci numbers. The `up_to` method returns a `FibLink` instance whose first element is the largest Fibonacci number that is less than or equal to positive integer n.

```python
class FibLink(Link):
    """Linked list of Fibonacci numbers."

    >>> ten = FibLink(2, FibLink(1)).up_to(10)
    >>> ten
    Link(8, Link(5, Link(3, Link(2, Link(1))))))
    >>> ten.up_to(1)
    Link(1)
    >>> six, thirteen = ten.up_to(6), ten.up_to(13)
    >>> six
    Link(5, Link(3, Link(2, Link(1))))
    >>> thirteen
    Link(13, Link(8, Link(5, Link(3, Link(2, Link(1))))))
    ""
    successor = self.first + self.rest
    @property
def successor():
def successor(self):
    return first + rest.first
    return self.first + self.rest.first

def up_to(n):
def up_to(self, n):
    while self.first > n:
        self = self.rest.first
        self = rest
        self.first = self.rest.first
    if self.first == n:
        return self
    elif self.first > n:
        return self.up_to(n)
        return self.rest.up_to(n)
    elif self.successor > n:
        return self
    else:
        return FibLink(self.successor(self), self).up_to(n)
        return FibLink(self.successor(self), self).up_to(n)
        return FibLink(self.successor(self), self.rest).up_to(n)
        return FibLink(self.successor(self), self.rest).up_to(n)
```

(c) (2 pt) Circle the Θ expression below that describes the number of calls made to `FibLink.up_to` when evaluating `FibLink(2, FibLink(1)).up_to(n)`. The constant $\phi$ is $\frac{1 + \sqrt{5}}{2} = 1.618...$

\[ \Theta(1) \quad \Theta(\log_\phi n) \quad \Theta(n) \quad \Theta(n^2) \quad \Theta(\phi^n) \]
(d) (2 pt) Alyssa P. Hacker remarks that Fibonacci numbers also form a basis. How many total calls to `FibLink.up_to` will be made while evaluating all the doctests of the `fib_basis` function below? Assume that `sum_to` and `FibLink` are implemented correctly. Write your answer in the box.

```
def fib_basis():
    """Fibonacci basis with caching."

    >>> r = fib_basis()
    >>> r(11)
    Link(8, Link(3))
    >>> r(23)
    Link(21, Link(2))
    >>> r(24)
    Link(21, Link(3))
    >>> r(45)
    Link(34, Link(8, Link(3)))
    ""
    fibs = FibLink(2, FibLink(1))
    def represent(n):
        nonlocal fibs
        fibs = fibs.up_to(n)
        return sum_to(n, fibs)
    return represent
```

(e) (4 pt) Implement `fib_sums`, a function that takes positive integer `n` and returns the number of ways that `n` can be expressed as a sum of unique Fibonacci numbers. Assume that `FibLink` is implemented correctly.

```
def fib_sums(n):
    """The number of ways `n` can be expressed as a sum of unique Fibonacci numbers.
    ""

    >>> fib_sums(9)  # 8+1, 5+3+1
    2
    >>> fib_sums(12) # 8+3+1
    1
    >>> fib_sums(13) # 13, 8+5, 8+3+2
    3
    ""
    def sums(n, fibs):
        """Ways `n` can be expressed as a sum of elements in `fibs`."

        if n == 0:
            return 1
        elif ________________________________________________________________:
            return 0
        a = ____________________________________________________________
        b = ____________________________________________________________
        return a + b
    return sums(n, FibLink(2, FibLink(1)).up_to(n))
```
6. (8 points) Sequels

Assume that the following table of movie ratings has been created.

```
correct output
<table>
<thead>
<tr>
<th>Judgment Day</th>
<th>Terminator</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Matrix</td>
<td>The Matrix</td>
</tr>
<tr>
<td>The Matrix</td>
<td>The Matrix</td>
</tr>
<tr>
<td>The Matrix</td>
<td>The Matrix</td>
</tr>
<tr>
<td>Toy Story 2</td>
<td>Toy Story</td>
</tr>
<tr>
<td>Toy Story 3</td>
<td>Toy Story</td>
</tr>
<tr>
<td>Terminator</td>
<td>Rise of the</td>
</tr>
<tr>
<td>Machines</td>
<td>Machines</td>
</tr>
</tbody>
</table>
```

```
correct output
Judgment Day
Terminator
The Matrix
Toy Story
Toy Story 2
Toy Story 3
```

The correct output table for both questions below happens to be the same. It appears above to the right for your reference. **Do not hard code your solution to work only with this table!** Your implementations should work correctly even if the contents of the `ratings` table were to change.

(a) **(2 pt)** Select the titles of all movies that have a rating greater than 7 in alphabetical order.

(b) **(6 pt)** Select the titles of all movies for which at least 2 other movies have the same rating. The results should appear in alphabetical order. Repeated results are acceptable. *You may only use the SQL features introduced in this course.*

```sql
with
    groups(name, score, n) as (  
        select __________, __________, __________ from ratings union  
        select __________, __________, __________ from groups, ratings  
        where ___________________________  
    )

drop group(n) as (  
    select title from _______________________________  
    where ___________________________________________  
    order by ____________________________  
);```
Appendix: Pair and nil Implementations
This page does not contain a question. These classes were originally defined in `scheme_reader.py`.

class Pair:
    """A pair has two instance attributes: first and second. For a Pair to be a well-formed list, second is either a well-formed list or nil. Some methods only apply to well-formed lists.

>>> s = Pair(1, Pair(2, nil))
>>> s
Pair(1, Pair(2, nil))
>>> print(s)
(1 2)
"""
    def __init__(self, first, second):
        self.first = first
        self.second = second
    def __repr__(self):
        return "Pair({0}, {1}).format(repr(self.first), repr(self.second))
    def __str__(self):
        s = "(" + str(self.first)
        second = self.second
        while isinstance(second, Pair):
            s += " " + str(second.first)
            second = second.second
        if second is not nil:
            s += ". " + str(second)
        return s + ")"

class nil:
    """The empty list""
    def __repr__(self):
        return "nil"
    def __str__(self):
        return "()"
    def __len__(self):
        return 0
    def __getitem__(self, k):
        if k < 0:
            raise IndexError("negative index into list")
        raise IndexError("list index out of bounds")
    def map(self, fn):
        return self

nil = nil()  # Assignment hides the nil class; there is only one instance
Scratch Paper
Scratch Paper
Scratch Paper
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.

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 first frame of the current environment.

Evaluation rule for or expressions:
1. Evaluate the header's expression.
2. If it is a true value, execute the suite, then skip the remaining clauses.

Evaluation rule for not expressions:
1. Evaluate the header's expression.
2. If it is a true value, execute the suite, then skip the remaining clauses.

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.

Evaluation rule for expressions:
1. Evaluate the subexpression on the left.
2. If the result is a true value, then the expression evaluates to True.
3. Otherwise, the expression evaluates to the value of the subexpression on the right.

Execution rule for and expressions:
1. Evaluate the subexpression on the left.
2. If the result is a false value, then the expression evaluates to False.
3. Otherwise, the expression evaluates to the value of the subexpression on the right.

High-order function: A function that takes a function as an argument or returns a function as a return value

Pure Functions
- abs(number):
  - 2
- pow(x, y):
  - 1024

Non-Pure Functions
- print(...)::
  - None

Nested def statements: Functions defined within other functions (no def within def) consists of one local frame, followed by the global frame
A function with formal parameters \( x \) and \( y \) that returns the value of \( x + y \)

\[
\text{square} = \lambda x, y: x \times y
\]

A function that returns a function

\[
def \text{make_adder}(n):
    \text{return adder}(k): k + n,
\]

A local def statement

\[
def \text{adder}(k):
    \text{return } k + n
\]

Can refer to names in the enclosing function

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 (global)
- The parent of a frame is the parent of the function called

A function's signature has all the information to create a local frame

Anatomy of a recursive function:
- The def statement header is similar to other functions
- Conditional statements check for base cases
- Base cases are evaluated without recursive calls
- Recursive cases are evaluated with recursive calls

\[
def \text{sum_digits}(n):
    \text{return } \text{floordiv}(n, 10) + \text{mod}(n, 10)
\]

Return the sum of the digits of positive integer \( n \)

If \( n \ge 10 \):
  \[\text{return } n\]
Else
  \[\text{return } \text{sum_digits}(\text{all but last}, \text{last}) + \text{last}\]

Program output:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

\[
def \text{get_fact}(n):
    \begin{cases}
    0 & \text{if } n = 0; \\
    1 & \text{return } n \text{ fact}(n-1)
    \end{cases}
\]

Return the factorial of \( n \)

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

From operator import \text{floor}(n, d)

def \text{divide_exact}(n, d):
  """Return the quotient and remainder of dividing \( n \) by \( d \)."
  \[
  \begin{align*}
  q & = \text{divide_exact}(2012, 10) \\
  r & = \text{divide_exact}(2012, 10) \\
  \end{align*}
  \]

Multiple return values, separated by commas

Multiple assignment to two names

\[
\text{square} = \lambda x: x \times x
\]

Both create a function with the same domain, range, and behavior.

Both functions have as their parent the environment in which they were defined.

Both bind that function to the name square.

Only the def statement gives the function an intrinsic name.

When a function is defined:
1. Create a function value: func <name> <formal parameters>
2. Its parent is the current frame.

\[
def \text{f_then_g}(f, g, n):
    \begin{cases}
    f(n) & \text{if } n == 0; \\
    g(n) & \text{return } 1
    \end{cases}
\]

3. Bind <name> to the function value in the current frame (which is the first frame of the current environment).

When a function is called:
1. Add a local frame, titled with the <name> of the function being called.
2. Copy the parent of the function to the local frame: [parent=label]
3. Bind the <formal parameters> to the arguments in the local frame
4. Execute the body of the function in the environment that starts with the local frame.
CS 61A Midterm 2 Study Guide — Page 1

**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

**Type dispatching:** Look up a cross-type implementation of an operation based on the types of its arguments
Type coercion: Look up a function for converting one type to another, then apply a type-specific implementation.

**List & dictionary mutation:**

```python
>>> a = [18]
>>> b = a
>>> a = b
True
True
>>> a.append(20)
>>> b.append(20)
True
a = b
True
>>> a == b
True
>>> a[10, 20]
[10, 20]
False
>>> nums = {'V': 1.0, 'V': 5, 'X': 10}
>>> nums['X']
10
>>> nums['V'] = 1
>>> nums['L'] = 50
>>> nums
('V': 1.0, 'L': 50, 'V': 5, 'I': 1)
>>> sum(nums.values())
66
>>> dict((3, 9), (4, 16), (5, 25))
{(3, 9): 4, 16, 5: 25}
>>> nums.get('A', 0)
0
>>> nums.get('W', 0)
5
>>> (x: x for x in range(3, 6))
((3, 5), 4, 6, 5, 25)
>>> original_suits = suits
>>> suits.pop('myriad')
'myriad'
>>> suits.remove('string!')
>>> suits.append('cup')
>>> suits.extend(['sword', 'club'])
>>> suits[2] = 'spade'  
>>> suits['coin', 'cup', 'spade', 'club']
{'coin': 'spade', 'spade', 'club'}
>>> suits[3, 2] = ['heart', 'diamond']
>>> suits['heart', 'diamond', 'spade', 'club']
['heart', 'diamond', 'spade', 'club']
```

**Identity:**

```python
lambda exp = exp
evaluates to True if both `exp` and `exp` evaluate to the same object
Equality:
lambda exp == exp
evaluates to True if both `exp` and `exp` evaluate to equal values
Identical objects are always equal values
You can copy a list by calling the list constructor or slicing the list from the beginning to the end.
```

**Constants:**

```python
>>> a = 18
>>> b = a
>>> a = b
True
>>> a.append(20)
>>> b.append(20)
True
a = b
True
>>> a == b
True
>>> a[10, 20]
[10, 20]
False
```

**Nesting:**

```python
>>> def overlap(a, b):
...   count = 0
...   for item in a:
...     if item in b:
...       count += 1
...       outer: length of a
...       inner: length of b
...     return count
...   return count
...   outer: length of a
...   inner: length of b
...   return count
```

**Lower-order terms:** The fastest-growing part of the computation dominates the total

```python
>>> def make_withdrawal(balance, amount, parent_frame):
...   if amount > balance:
...     return 'No funds'
...   balance = balance - amount
...   return balance
```

**Strings as sequences:**

```python
>>> city = 'Berkeley'
>>> len(city)
6
>>> 'k' in 'Where's Waldo?'
True
>>> 'k' in 'Where's Waldo?'
False
>>> 234 in [1, 2, 3, 4]
False
>>> 'w' in 'w' and 'b' in 'b'
True
>>> [2, 'w'] in [[1, 2, 3], 2, 2, 2, False]
False
```

**Membership:**

```python
>>> if item in b:
...   count += 1
...   outer: length of a
...   inner: length of b
...   return count
...   outer: length of a
...   inner: length of b
...   return count
```

**Slicing:**

```python
>>> [2, 3, 4] in [[1, 2, 3], 2, 2, 2, False]
False
```

**Enumerating a list:**

```python
>>> list(range(3))
[0, 1, 2]
```

**Unpacking a list:**

```python
>>> list(range(3))
[0, 1, 2]
```

**Type coercion:**

```python
>>> if x == y:
...   return x
...   same_count = same_count + 1
```

**For each element in a fixed-length sequence:**

```python
>>> if x == y:
...   return x
...   same_count = same_count + 1
```

**For each element in a fixed-length sequence:**

```python
>>> if x == y:
...   return x
...   same_count = same_count + 1
```

**For each element in a fixed-length sequence:**

```python
>>> if x == y:
...   return x
...   same_count = same_count + 1
```

**A sequence of fixed-length sequences:**

```python
>>> a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```
Linked list data abstraction:
```
def link(first, rest):
    return [first, rest]
def first(s):
    return s[0]
def rest(s):
    return s[1]
def len_link(s):
    x = 0
    while s != empty:
        s, x = rest(s), x+1
    return x
def getitem_link(s, i):
    while i > 0:
        s, i = rest(s), i - 1
    return first(s)
def extend(s, t):
    assert is_link(s) and is_link(t)
    if s == empty:
        return t
    else:
        return link(first(s), extend(rest(s), t))
def apply_to_all_link(f, s):
    if s == empty:
        return s
    else:
        return link(f(first(s)), apply_to_all_link(f, rest(s)))
```

The result of calling `repr` on a value is what Python prints in an interactive session (CS 61A Midterm 2 Study Guide).

The result of calling `str` on a value is what Python prints using the `print` function:
```
>>> 12e12
12000000000000.0
>>> print(today)
2014-10-13
>>>
```

The 0-indexed element of the pair is the first element of the linked list. The 1-indexed element of the pair is the rest of the linked list.

A linked list is a pair
```
0 1 2 3 4
```

Sequence abstraction special names:
```
__getitem__ Element selection []
__len__ Built-in len function
```

Yes, this call is recursive
```
def __init__(self, first, rest=None):
    self.first = first
    self.rest = rest
def getitem(self, i):
    if i == 0:
        return self.first
    else:
        return self.rest[i-1]
def __len__(self):
    return 1 + len(self.rest)
```

Sequence
```
A linked list is a pair
```

Memoization:
```
def memoized(f):
    cache = {}
    def memoize(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
```

Assignments with a dot expression on their left-hand side affect attributes for the object of that dot expression. If the object is a class, then assignment sets a class attribute value.

Python object system:

Idea: All bank accounts have a `balance` and an account `holder`;
The `Account` class should add those attributes to each of its instances.
```
A new instance is created by calling a class
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
When a class is called:
1. A new instance of that class is created;
2. The `__init__` method of the class is called with the new object as its first argument (name `self`), along with any additional arguments provided in the call expression.
```
```
class Account:
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder

    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
    def withdraw(self, amount):
        if amount > self.balance:
            return 'Insufficient Funds'
        self.balance = self.balance - amount
        return self.balance
```

Method invocation:
```
Account.deposit(a, 5)
```

The `<expression>.<name>` can be any valid Python expression. The `<name>` must be a simple name.

Evaluates to the value of the attribute looked up by `<name>` in the object that is the value of the `<expression>`.

To evaluate a dot expression:
1. Evaluate the `<expression>` to the left of the dot, which yields the object of the dot expression.
2. Look up the `.<name>` part (method invocation) 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.

Assignment statements with a dot expression on their left-hand side affect attributes for the object of that dot expression. If the object is a class, then assignment sets a class attribute value.
```
Account attributes
| balance: 0 | holder: 'Jim' |
| balance: 0 | holder: 'Tom' |
```

Class `CheckingAccount`:
```
class CheckingAccount(Account):
    """A bank account that charges for withdrawals."
    withdraw_fee = 1
    interest = 0.01
    def withdraw(self, amount):
        return Account.withdraw(self, amount + self.withdraw_fee)
        or
        super().withdraw(amount + self.withdraw_fee)
```
Exceptions are raised with a raise statement.

```python
try: ...
except ... as name:
...
```
The `try` suite is executed first. If, during the course of the `try` suite, an exception is raised that is not handled otherwise, and if the class of the exception inherits from `Exception`, then the `except` suite is executed, with the bound to the exception.

```python
class A(BaseException):
    pass
```

```python
for name in expression:
  <suite>
```
1. Evaluate the header `expression`, which yields an iterable object.
2. For each element in that sequence, in order:
   A. Bind `name` to that element in the first frame of the current environment.
   B. Execute the `suite`.
An iterable object has a method `__iter__` that returns an iterator.

```python
def __iter__(self):
  return items.__iter__()
```

A stream is a linked list, but the rest of the list is computed on demand. Once created, Streams and Rlists can be used interchangeably using `first` and `rest`.

```python
class Stream:
  def __iter__(self):
    return StreamEmpty.
  def __next__(self):
    return StreamEmpty.
```

A generator is an iterator backed by a generator function. Each time a generator function is called, it returns a generator.

```python
>>> try:
...     x = 1/0
... except ZeroDivisionError as e:
...     print('handling a', e, 'type(e):', type(e))
...     x = 8
>>> handling a ZeroDivisionError type(e): <class 'ZeroDivisionError'>
>>> x
8
```

```python
class Filter:
  def __iter__(self):
    return items.__next__()
```

```python
A row has a value for each column
```
A Scheme list is written as elements in parentheses:

- Each `<element>` can be a combination or atom (primitive).
- A combination can be a `lookup` of tokens.
- Recursive calls: `Eval(operand, operands)` of call expressions
- `Eval(sub-expressions) of special forms`

**Base cases:**
- `Primitive values (numbers)`
- `Look up values bound to symbols`

**Recursive calls:**
- `Eval(operator, operands)` of call expressions
- `Eval(sub-expressions)` of special forms

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

Each call to `scheme_read` consumes the input tokens for exactly one expression.

**Base case:** symbols and numbers

**Recurrsive call:** `scheme_read` sub-expressions and combine them

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

- **Lines**
  - `Parser` (scheme_reader.py)
  - `Evaluator` (scalc.py)
  - `expression` reads `value`

- **Line**
  - `('(+ 2 3)')` evaluates to `5`
  - `('(+ 2 3)')` evaluates to `5`
  - `('(+ 2 3)')` evaluates to `5`
  - `('(+ 2 3)')` evaluates to `5`

A Scheme list is a pair in which the second element is `nil` or a Scheme list.

- A linked Scheme list consists of elements in parentheses.

- A dotted list has an arbitrary value for the second element of the last pair. Dotted lists may not be well-formed lists.

Examples of Scheme expressions:

- `(length-iter k)`
- `(list 'a 'b)`
- `(lambda (x) (+ x 4))`
- `(if (< x 0) (- x) x))`
- `(cons 1 (cons 2 (cons 3 (cons 4 nil))))`
- `(define x (cons 1 2))`
- `(define b 2)`
- `(define a 1)`
- `(define (plus4 x) (+ x 4))`

Expressions 2 & 3 (consequent & alternative) in a tail context

- Recursive call: process multiple lines
- Base cases: created a new environment each time a user-defined procedure is applied
- Requires an environment for name lookup
- Apply base cases: built-in primitive procedures
- Recursive calls: `Eval(body)` of user-defined procedures

To apply a user-defined procedure, create a new frame in which all 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.

```
(define (f s) (if (null? s) '(3 (cons (car s) (f (cdr s))))
(f (list 1 2))))
```

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 (length s) ...
(if (null? s) 0
  (+ 1 (length (cdr s)))))
```

A recursive call is a tail call.

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

Not a tail call

Images for expressions:

- `Expression Tree`
- `Calcualtor Expression`
- `Expression Calculator Expression`
- `Pair (+, Pair(2, Pair(2, nil)))`
- `Pair((lambda (x y z) (+ x y (square z))) 1 2 3)`