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 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>Q. 5</th>
<th>Q. 6</th>
<th>Q. 7</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>/12</td>
<td>/16</td>
<td>/8</td>
<td>/7</td>
<td>/8</td>
<td>/16</td>
<td>/13</td>
<td>/80</td>
</tr>
</tbody>
</table>
1. (12 points) We Are Binary Tree Huggers

This problem makes use of the Tree class from lecture; its definition is contained in the Midterm 2 Study Guide, attached to the end of this exam.

(a) The depth of a tree is defined as the number of nodes encountered in the longest path from the root to a leaf. Complete the function definition below to compute the depth of a binary tree.

```python
def depth(tree):
    """Compute the depth of a binary tree."

    >>> t = Tree(6, Tree(2, Tree(1)), Tree(7))
    >>> depth(t)
    3
    >>> t.left.right = Tree(4, Tree(3), Tree(5))
    >>> depth(t)
    4
    """

    if _____________________________________________________________:
        return 1 + max(___________________________________________,
                        _____________________________________________)
```

(b) A binary tree is balanced if for every node, the depth of its left subtree differs by at most 1 from the depth of its right subtree. Fill in the definition of the is_balanced function below to determine whether or not a binary tree is balanced. You may assume that the depth function works correctly for this part.

```python
def is_balanced(tree):
    """Determine whether or not a binary tree is balanced."

    >>> t = Tree(6, Tree(2, Tree(1)), Tree(7))
    >>> is_balanced(t)
    True
    >>> t.left.right = Tree(4, Tree(3), Tree(5))
    >>> is_balanced(t)
    False
    """
```
(c) For the following class definition, cross out any incorrect or unnecessary lines in the following code so that the doctests pass. **Do not cross out class declarations, doctests, or docstrings.** You can cross out anything else, including method declarations, and your final code should be as compact as possible. **Make sure to cross out the entire line for anything you wish to remove.** You may assume that the `depth` and `is_balanced` functions are defined correctly in the global environment.

```python
class STree(Tree):
    """A smart tree that knows its depth and whether or not it is balanced."

    >>> s = STree(6, STree(2, STree(1)), STree(7))
    >>> s.depth
    3
    >>> s.is_balanced
    True
    >>> s.left.right = STree(4, STree(3), STree(5))
    >>> s.depth
    4
    >>> s.is_balanced
    False
    ""

def __init__(self, entry, left=None, right=None):
    Tree.__init__(entry, left, right)
    self.entry = entry
    self.left = left
    self.right = right
    self.depth = depth(self)
    self.is_balanced = is_balanced(self)

@property
def depth(self):
    return depth(self)

@property
def is_balanced(self):
    return is_balanced(self)
```
2. (16 points) Binary Tree Huggers are Environmentalists

(a) (7 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 need only show the final state of each frame. You may not need to use all of the spaces or frames. You may draw objects that are created but are not accessible from the environment, if you wish. Make sure to reflect every call to a user-defined function in the environment diagram.

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
x = [0, 1]
y = list(map(lambda x: 2*x, x + [2]))
y[2] = y
z = y[:]
z[2][2] = z
```

---

```
x = [0, 1]
y = list(map(lambda x: 2*x, x + [2]))
y[2] = y
z = y[:]
z[2][2] = z
```

---

Global frame

```
x
```

```
list
```

```
0 1
```

```
Return Value
```

```
Return Value
```

```
Return Value
```

```
Return Value
```

```
Return Value
```
(b) (9 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 need only show the final state of each frame. You may not need to use all of the spaces or frames. You may draw objects that are created but are not accessible from the environment, if you wish. Make sure to reflect every call to a user-defined function in the environment diagram.

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 eval(expr):
    if type(expr) in (int, float):
        return expr
    procedure = expr[0]
    args, i = [], 1
    while i < len(expr):
        args.append(eval(expr[i]))
        i += 1
    return procedure(*args)
expr = [lambda x: [x * x] + expr, 4]
result = eval(expr)
```

```
Global frame
func eval(expr)
```

```
Return Value
```

```
Return Value
```

```
Return Value
```

```
Return Value
```
3. (8 points) We Recurse at Hard Problems

The following are the first six rows of Pascal’s triangle:

\[
\begin{array}{cccc}
1 \\
1 & 1 \\
1 & 2 & 1 \\
1 & 3 & 3 & 1 \\
1 & 4 & 6 & 4 & 1 \\
1 & 5 & 10 & 10 & 5 & 1 \\
\end{array}
\]

The first and last element in each row is 1, and each of the other elements is equal to the sum of the element above it and to the left and the element above it and to the right. For example, the third element in the last row is \(4 + 6 = 10\), since 4 and 6 are the elements above it and to the left and right.

(a) Define a function \texttt{pascal} that takes a row index \(n\) and an element index \(k\) as arguments and computes the \(k\)th element in row \(n\), with indexing beginning at 0 for both \(n\) and \(k\). \textbf{Do not compute factorial or any other combinatorial expression as part of your solution.}

```python
def pascal(n, k):
    """Compute the kth element of the nth row in Pascal’s triangle."
    >>> pascal(5, 0)
    1
    >>> pascal(5, 2)
    10
```

(b) Fill in the `pascal_gen` function below, which returns an iterator over the elements in the \( n \)th row of Pascal’s triangle. Your solution should be self-contained: you may **not** use the `pascal` function defined in part (a).

```python
def pascal_gen(n):
    """Return an iterator over all the elements in the \( n \)th row of Pascal’s triangle.

    >>> list(pascal_gen(5))
    [1, 5, 10, 10, 5, 1]
    ""

    if ____________________________:
        last = 0
        for num in ____________________________:
            yield ______________________________

            ______________________________

        yield 1
```
4. (7 points)  We are Functionally Lazy

(a) Fill in the function below to match its docstring description, so that all doctests pass.

```python
def squares(num):
    '''Return the square of num and a function to compute subsequent squares.

    >>> s, f = squares(1)
    >>> s
    1
    >>> s, f = f()
    >>> s
    4
    >>> s, f = f()
    >>> s
    9
    '''
    squared = num * num
    func = ________________________________
    return ________________________________
```

(b) Fill in the function below to match its docstring description, so that all doctests pass.

```python
def make_countdown(start):
    '''Return a function that will count down from start to 1, returning the next value each time it is called, and returning 'GO!' when it is done.

    >>> countdown = make_countdown(3)
    >>> countdown()
    3
    >>> countdown()
    2
    >>> countdown()
    1
    >>> countdown()
    'GO!'
    >>> countdown()
    'GO!'
    '''
```
5. (8 points) We are Objectively Lazy

Suppose we wish to define a new lazily evaluated list type called LazyList. A LazyList does not hold elements directly; instead, it holds 0-argument functions to compute each element. The first time an element is accessed, the LazyList calls the stored function to compute that element. Subsequent accesses to the same element do not call the stored function. See the docstring for LazyList for examples of how to use it.

(a) Fill in the class definition of LazyList below to match its docstring description, so that all doctests pass.

```python
class LazyList(object):
    """A lazy list that stores functions to compute an element. Calls the appropriate function on first access to an element; never calls an element's function more than once."

    >>> def compute_number(num):
    ...     print('computing', num)
    ...     return num
    ...
    >>> s = LazyList()
    >>> s.append(lambda: compute_number(1))
    >>> s.append(lambda: compute_number(2))
    >>> s.append(lambda: compute_number(3))
    >>> s[1]
    computing 2
    2
    >>> s[1]
    2
    >>> s[0]
    computing 1
    1
    >>> for item in s: print(item)
    1
    2
    computing 3
    3
    """
    def __init__(self):
        self._list = []
        self._computed_indices = set()

    def append(self, item):
        ________________________________

    def __getitem__(self, index):
        if ________________________________:
            self._computed_indices.add(index)
        ________________________________
        return self._list[index]
```
def __iter__(self):
    for ________________________________________________________:
    __________________________________________________________

(b) Assuming a correct definition of LazyList, what value would be bound to result after executing the following code? Circle the value below, or “other” if the value is not one of the choices provided. (Hint: Draw the environment diagram for mystery and the functions defined within it.)

def mystery(n):
    s = LazyList()
    i = 0
    while i < n:
        s.append(lambda: i)
        i += 1
    return s

result = mystery(4)[1]

0 1 2 3 4 other
6. (16 points) We Love to Scheme; Muahahaha!

(a) Assume that you have started the Scheme interpreter and defined the following procedures:

```scheme
(define x (lambda (y)
   (if (= y 0)
       1
       (+ (x (- y 1)) y))))

(define y (mu (x)
   (if (= x 0)
       1
       (- (y (- x 1)) x))))

(define (z x y) (x y))
(define (r x y r) (x r))
```

For each of the following expressions, write the value to which it evaluates. If the value is a function value, write FUNCTION. If evaluation causes an error, write ERROR. If evaluation would run forever, write FOREVER. Otherwise, write the resulting value as the interactive interpreter would display it. The first two rows have been provided as examples:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Evaluates to</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+ 1 4)</td>
<td>5</td>
</tr>
<tr>
<td>(+ 1 car)</td>
<td>Error</td>
</tr>
<tr>
<td>(cons 1 (cons 2 3))</td>
<td></td>
</tr>
<tr>
<td>(cdr '(1 (2) 3))</td>
<td></td>
</tr>
<tr>
<td>(z car (list 1 (2) 3))</td>
<td></td>
</tr>
<tr>
<td>(z z z)</td>
<td></td>
</tr>
<tr>
<td>(z x 3)</td>
<td></td>
</tr>
<tr>
<td>(z y 3)</td>
<td></td>
</tr>
<tr>
<td>(r x y 3)</td>
<td></td>
</tr>
<tr>
<td>(r y x 3)</td>
<td></td>
</tr>
</tbody>
</table>
(b) Write a Scheme function `insert` that creates a new list that would result from inserting an item into an existing list at the given index. Assume that the given index is between 0 and the length of the original list, inclusive.

```scheme
(define (insert lst item index)
  (if _______________________________________________________________
      _______________________________________________________________
      _______________________________________________________________
    _______________________________________________________________
    _______________________________________________________________
))
```

(c) Suppose a tree abstract data type is defined as follows:

```scheme
;;; An empty tree.
(define empty-tree nil)

;;; Determine if a tree is empty.
(define (empty? tree) (null? tree))

;;; Construct a tree from an element and left and right subtrees.
(define (tree elem left right) (list elem left right))

;;; Retrieve the element stored at the given tree node.
(define (elem tree) (car tree))

;;; Retrieve the left subtree of a tree.
(define (left tree) (car (cdr tree)))

;;; Retrieve the right subtree of a tree.
(define (right tree) (car (cdr (cdr tree))))
```

Fill in the `contains` procedure below, which determines whether or not a number is contained in a set represented by the tree data structure above.

```scheme
(define (contains tree num)
  (if _______________________________________________________________
    false
    (if _______________________________________________________________
      true
      (if _______________________________________________________________
        _______________________________________________________________
        _______________________________________________________________
)))))
```
7. (13 points) Our Schemes are Logical, and Our Logic is Schemy

(a) Assume that you have started the Logic interpreter and defined the following relations:

\[
\begin{align*}
&\text{(fact (append () ?x ?x))} \\
&\text{(fact (append (?a . ?r) ?s (?a . ?t)) (append ?r ?s ?t))} \\
&\text{(fact (foo () () ()))} \\
&\text{(fact (foo (?a . ?r) (?b . ?s) ((?a ?b) . ?t)) (foo ?r ?s ?t))} \\
&\text{(fact (bar (?a . ?r) ?s) (append ?r (?a) ?s))} \\
&\text{(fact (baz ?rel () ()))} \\
\end{align*}
\]

For each of the following expressions, write the output that the interactive Logic interpreter would produce. The first two rows have been provided as examples:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>(query (append (1 2) 3 (1 2 3)))</td>
<td>Failed.</td>
</tr>
<tr>
<td>(query (append (1 2) (3 4) ?what))</td>
<td>Success!</td>
</tr>
<tr>
<td></td>
<td>what: (1 2 3 4)</td>
</tr>
<tr>
<td>(query (foo 1 3 (1 3)))</td>
<td></td>
</tr>
<tr>
<td>(query (foo (1) (3) (1 3)))</td>
<td></td>
</tr>
<tr>
<td>(query (foo (1 2) (3 4) ?what))</td>
<td></td>
</tr>
<tr>
<td>(query (bar () ()))</td>
<td></td>
</tr>
<tr>
<td>(query (bar (1 2 3 4) ?what))</td>
<td></td>
</tr>
<tr>
<td>(query (baz bar ((1 2 3) (4 5) (6)) ?what))</td>
<td></td>
</tr>
</tbody>
</table>
(b) Write a relation `sorted` that is true if the given list is sorted in increasing order. Assume that you have a \(<=\) relation that relates two items if the first is less than or equal to the second. Here are some sample facts and queries:

```prolog
logic> (fact (<= a a))
logic> (fact (<= a b))
logic> (fact (<= a c))
logic> (fact (<= b b))
logic> (fact (<= b c))
logic> (fact (<= c c))
logic> (query (sorted ()))
Success!
logic> (query (sorted (a b b c)))
Success!
logic> (query (sorted (b a c)))
Failed.
```

(c) Fill in the `all<=all` relation below, which relates two lists if every element in the first list is \(<=\) every element in the second list. You may use the `<=all` relation defined below. Here are some sample queries:

```prolog
logic> (query (all<=all (a b c) (a b c)))
Failed.
logic> (query (all<=all (a b) (b b c)))
Success!

(fact (<=all ?x ()))
(fact (<=all ?x (?a . ?r))
  (<= ?x ?a)
  (<=all ?x ?r))

(fact (all<=all () ?x))

(fact (all<=all -------------------------------))
  (------------------------------------------)
  (------------------------------------------)
  (all<=all -------------------------------))
```
A name is bound to a value in a frame, there is at most one binding per name.

### Evaluation rule for call expressions:
1. Evaluate the header's expression.
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.

### 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 or 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 subexpression <right>.

### 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 subexpression <right>.

### Evaluation rule for not expressions:
1. Evaluate the operand subexpression.
2. Evaluate \( \neg x \).

### Higher-order function:
A function that takes a function as an argument or returns a function.

### Nested def statements:
Functions defined within other function bodies are bound to names in the local frame.
def make_adder(n):
    """Return a function that takes one argument k and returns k + n.
    >>> add_three = ...
    an answer without calling itself
2. Recursive case(s), where the function calls itself
    as part of the computation
    """

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

make_adder(1)(2)

make_adder(1) ( 2  )

An expression that evaluates to a function value

An expression that evaluates to any value

def square(x):
    return x * x

square = lambda x, y: x * y

A function with formal parameters x and y
and body "return x * y"

Must be a single expression

A function that returns a function

>>> add_three = make_adder(3)

def square(lambda x, y: x * y):
    return square(x) + square(y)

square = lambda x: x ** 2

Facts about print
• Non-pure function
• Returns None
• Multiple arguments are printed with a space between them

>>> print(4, 2)
4 2

How to find the square root of 2?

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)

def iter_improve(update, done, guess, max_updates):
    """Iteratively improve guess with update until done returns a true value.
    >>> iter_improve(golden_update, golden_test)
    1.618033988749895
    """
    k = 0
    while not done(guess) and k < max_updates:
        k += 1
        return guess

    def newton_update(f):
        """Return an update function for f using Newton's method.
        """
        df = lambda x: f(x) / approx_derivative(f, x)
        return df

    def approx_derivative(f, x, delta=1e-5):
        """Return an approximation to the derivative of f at x.
        """
        df = (f(x + delta) - f(x)) / delta
        return df

    def find_zero(f, guess):
        """Return a guess of a zero of the function f, near guess.
        """
        from math import sin
        >>> find_zero(lambda y: sin(y), 3)
        3.141592653589793
        return iter_improve(newton_update(f), lambda x: f(x) == 0, guess)

A function is recursive if the body calls the function itself, either directly or indirectly
Recursive functions have two important components:
1. Base cases(s), where the function directly computes an answer without calling itself
2. Recursive case(s), where the function calls itself as part of the computation

def factorial(n):
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)

factorial(4)

A function's signature has all the information to create a local frame
```python
1. Evaluate the header <expression>, which must yield an iterable value.
2. For each element in that sequence, in order:
   A. Bind <name> to that element in the local environment.
   B. Execute the <suite>.

A range is a sequence of consecutive integers.*

...,-5,-4,-3,-2,-1,0,1,2,3,4,5,...

```
A class statement creates a new class and binds that class to `<name>` in the first frame of the current environment.

Statements in the `class` create attributes of the class.

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. If not, `<name>` is looked up in the class, which yields a class attribute value.
3. 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.

### Example

```python
>>> jim_account = Account('Jim')
>>> jim_account.balance
100
```

A simple container implemented using two accessor methods

```python
def container(contents):
    def get():
        return contents
    def put(value):
        contents = value
        return get()
def put(value):
    nonlocal contents
    contents = value
    return get()
```

### Example

```python
class ComplexRlist(object):
    def _init_(self, real, imag):
        self.real = real
        self.imag = imag
    def add_complex_and_rational(x, y):
        return ComplexRlist(x.real + y.real, x.imag + y.imag)
```

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

A mutable Rlist implementation using message passing

```python
def mutable_rlist():
    contents = empty_rlist
    def dispatch(message, value=None):
        local contents
        if message == 'len':
            return len_rlist(contents)
        elif message == 'getitem':
            return getitem_rlist(contents, value)
        elif message == 'push_first':
            contents = make_rlist(value, contents)
        return self
    def set_value(self, new, amount):
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return new
    return dispatch
```

A bank account implemented using dispatch dictionaries

```python
def account(balance):
    if amount < 0:
        return 'Insufficient funds'
    elif amount > 0:
        return return dispatch(balance)
    return deposit(amount):
        return return dispatch(balance)
    return withdraw(amount):
        if amount > balance:
            return 'Insufficient funds'
        new_balance = balance - amount
        return dispatch = (balance, 'withdraw', 'deposit', deposit)
```

A simple container implemented using two accessor methods
The interface for sets:
- Membership testing: Is a value an element of a set?
- Adjunction: Return a set with all elements in s and value v.
- Union: Return a set with all elements in set1 and set2.
- Intersection: Return a set with any elements in set1 and set2.

### Table
<table>
<thead>
<tr>
<th>Operation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Membership</td>
<td>Is a value an element of a set?</td>
</tr>
<tr>
<td>Adjunction</td>
<td>Return a set with all elements in s and value v.</td>
</tr>
<tr>
<td>Union</td>
<td>Return a set with all elements in set1 and set2.</td>
</tr>
<tr>
<td>Intersection</td>
<td>Return a set with any elements in set1 and set2.</td>
</tr>
</tbody>
</table>

### Exceptions
Exceptions are raised with a raise statement.

```python
raise <expression>
```

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

### Try and Except

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

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.

### Streams
A stream is a sequence of values that are lazily computed recursive lists.

```python
class Stream(Rlist):
  def __init__(self, first, compute_rest=fnlisma: Stream.empty):  # constructor
    if not callable(compute_rest):
      raise TypeError("compute_rest must be callable")
    self.first = first
    self.compute_rest = compute_rest
    self._rest = None

  def __repr__(self):
    return "Stream({0}, <...>)".format(repr(self.first))

  def __iter__(self):
    while self:  # while loop
      yield self.first
      self = self.compute_rest()

  def empty():  # class method
    return Stream.empty()

  def append(self, obj):  # instance method
    if callable(self._compute_rest):
      self._compute_rest = lambda: Stream.append(self, obj)
    else:
      self._compute_rest = fnlisma: Stream.append(self, obj)  # bound lambda

  def extend(self, seq):  # instance method
    if callable(self._compute_rest):
      self._compute_rest = lambda: Stream.extend(self, seq)
    else:
      self._compute_rest = fnlisma: Stream.extend(self, seq)  # bound lambda

  def __len__(self):  # property
    return len_not_supported_on_Stream()

  def __contains__(self, elem):  # property
    return elem in self

  def index(self, elem):  # property
    return IndexNotSupportedInStream()

  def count(self, elem):  # property
    return CountNotSupportedInStream()

  def map_stream(self, fn):  # instance method
    return MapStream(self, fn)

  def filter_stream(self, fn):  # instance method
    return FilterStream(self, fn)

  def scanl_stream(self, fn):  # instance method
    return ScanlStream(self, fn)

  def scanr_stream(self, fn):  # instance method
    return ScanrStream(self, fn)

  def foldl_stream(self, fn, init):  # instance method
    return FoldlStream(self, fn, init)

  def foldr_stream(self, fn, init):  # instance method
    return FoldrStream(self, fn, init)

  def reverse(self):  # instance method
    return ReverseStream(self)

def MapStream(s, fn):
  return Stream.map_stream(fn, s)

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

def ScanlStream(s, fn):
  return Stream.scanl_stream(fn, s)

def ScanrStream(s, fn):
  return Stream.scanr_stream(fn, s)

def FoldlStream(s, fn, init):
  return Stream.foldl_stream(fn, init, s)

def FoldrStream(s, fn, init):
  return Stream.foldr_stream(fn, init, s)

def ReverseStream(s):
  return Stream.reverse()

def first(s):
  return s.first

def rest(s):
  return s.compute_rest()

def prime(pos_stream):
  def not_divisible(x):
    return x % pos_stream.first != 0

  def compute_rest():
    return primes(not_divisible, pos_stream.rest() if not_divisible(pos_stream.first) else

  return Stream(primes, first, compute_rest)
```

### Examples

Example of using a stream to compute the Fibonacci sequence:

```python
fib = Stream(1, Stream(1))
for n in range(10):
    print(fib)  # Output: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55
```

### Logic
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
log\(\langle\text{fact}\rangle\) (parent delano herbert)
log\(\langle\text{fact}\rangle\) (parent abraham barack)
log\(\langle\text{fact}\rangle\) (parent abraham clinton)
log\(\langle\text{fact}\rangle\) (parent fillmore herbert)
log\(\langle\text{fact}\rangle\) (parent fillmore grover)
log\(\langle\text{fact}\rangle\) (parent eisenhower fillmore)
```

### Variables and Expressions
Variables can refer to atoms or relations in queries.

```python
log\(\langle\text{query}\rangle\) (parent abraham ?child)
Success!
child: barack
child: clinton
log\(\langle\text{query}\rangle\) (dog (name abraham) (color white))
Success!
color: white
log\(\langle\text{query}\rangle\) (parent ?child fillmore)
Success!
child: abraham
child: delano
child: grover
```

A fact is recursive if the same relation is mentioned in a hypothesis and the conclusion.

```python
log\(\langle\text{fact}\rangle\) (ancestor ?a ?b)
log\(\langle\text{fact}\rangle\) (ancestor ?a ?b)
log\(\langle\text{query}\rangle\) (ancestor ?a ?b)
log\(\langle\text{query}\rangle\) (ancestor ?a ?b)
```

### Lists
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
log\(\langle\text{fact}\rangle\) (append-to-form (?x ?y))
log\(\langle\text{fact}\rangle\) (append-to-form (?a ?r) ?y (?a ?z))
log\(\langle\text{fact}\rangle\) (append-to-form ?r ?y)
```

### Classes
A class is a Scheme list.

```python
class Letters(object):
  """An iterator over letters."""
  def __init__(self):
    self._current = 'a'
  def __next__(self):
    if self._current == 'z':
      raise StopIteration
      self._current += 1
    return self._current

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

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

### Functions
- A generator is an iterator backed by a generator function.
- When a generator function is called, it returns a generator.
A basic interpreter has two parts: a parser and an evaluator.

### Scheme Programs

Scheme programs consist of expressions, which can be:
- Primitive expressions:
  - 
  - Numbers are self-evaluating; symbols are bound to values.
- Call expressions have an operator and 0 or more operands.
  - Numbers are self-evaluating; symbols are bound to values.
  - New procedures:
  - Binding names:
  - Combinations:
    - (quotient 10 2)
    - New procedures:
    - Built-in primitive procedures
    - Apply (operator, arguments)
  - Evaluating expressions of special forms

### Expressions

- Expressions can be:
  - Numbers
  - Symbols
  - Combinations:
    - (define (f s) (if (< s 0) (- s) (+ s)))
    - New procedures:
    - Binding names:
    - Combinations:
      - (quotient 10 2)
      - New procedures:
      - Built-in primitive procedures
      - Apply (operator, arguments)
    - Evaluating expressions of special forms

### Syntax Analysis

- Syntax analysis identifies the hierarchical structure of an expression to the expression itself.
- Evaluator uses this structure to evaluate the expression.

### Expression Trees

- Expression trees are used to represent the structure of an expression.
- Trees consist of nodes, each representing a part of the expression.
- The leaves of the tree represent the values of the expression.
- The nodes represent the operations performed on the values.

### Scheme List

- A Scheme list is written as elements in parentheses.
- A Scheme list can be an expression, a number or a Scheme list.
- Two equivalent expressions:
  - (define (even? x) (= (remainder x 2) 0))
  - New procedures:
  - Built-in primitive procedures
  - Apply (operator, arguments)
  - Evaluating expressions of special forms

### Scheme Interpreter

- The Scheme interpreter takes a sequence of expressions and evaluates them.
- The interpreter uses a stack to store the values of the expressions.
- The interpreter supports recursive procedures.

### Scheme List

- A Scheme list is written as elements in parentheses.
- Each element in a Scheme list can be a combination or atom (primitive).
- A Scheme list can be an expression, a number or a Scheme list.
- New procedures:
  - Built-in primitive procedures
  - Apply (operator, arguments)
  - Evaluating expressions of special forms

### Scheme Reader

- The Scheme reader parses the expressions and builds an abstract syntax tree.
- The parser checks for malformed tokens and builds the abstract syntax tree.
- New procedures:
  - Built-in primitive procedures
  - Apply (operator, arguments)
  - Evaluating expressions of special forms

### Scheme Compiler

- The Scheme compiler translates the abstract syntax tree into machine code.
- New procedures:
  - Built-in primitive procedures
  - Apply (operator, arguments)
  - Evaluating expressions of special forms

### Scheme Execution

- The Scheme interpreter executes the machine code to evaluate the expressions.
- The interpreter supports recursive procedures.

### Scheme Environment

- A Scheme environment is a data structure that associates names with values.
- The environment is used to lookup values in the expressions.
- New procedures:
  - Built-in primitive procedures
  - Apply (operator, arguments)
  - Evaluating expressions of special forms

### Scheme Procedure

- A Scheme procedure is a function that takes arguments and returns a value.
- New procedures:
  - Built-in primitive procedures
  - Apply (operator, arguments)
  - Evaluating expressions of special forms

### Scheme Frame

- A Scheme frame is a data structure that associates names with values.
- The frame is used to store the arguments and values of procedures.
- New procedures:
  - Built-in primitive procedures
  - Apply (operator, arguments)
  - Evaluating expressions of special forms

### Scheme Pair

- A Scheme pair is a data structure that consists of two elements.
- New procedures:
  - Built-in primitive procedures
  - Apply (operator, arguments)
  - Evaluating expressions of special forms

### Scheme List

- A Scheme list is written as elements in parentheses.
- Each element in a Scheme list can be a combination or atom (primitive).
- A Scheme list can be an expression, a number or a Scheme list.
- New procedures:
  - Built-in primitive procedures
  - Apply (operator, arguments)
  - Evaluating expressions of special forms

### Scheme Interpreter

- The Scheme interpreter takes a sequence of expressions and evaluates them.
- The interpreter uses a stack to store the values of the expressions.
- The interpreter supports recursive procedures.

### Scheme Compiler

- The Scheme compiler translates the abstract syntax tree into machine code.
- New procedures:
  - Built-in primitive procedures
  - Apply (operator, arguments)
  - Evaluating expressions of special forms

### Scheme Execution

- The Scheme interpreter executes the machine code to evaluate the expressions.
- The interpreter supports recursive procedures.

### Scheme Environment

- A Scheme environment is a data structure that associates names with values.
- The environment is used to lookup values in the expressions.
- New procedures:
  - Built-in primitive procedures
  - Apply (operator, arguments)
  - Evaluating expressions of special forms

### Scheme Procedure

- A Scheme procedure is a function that takes arguments and returns a value.
- New procedures:
  - Built-in primitive procedures
  - Apply (operator, arguments)
  - Evaluating expressions of special forms

### Scheme Frame

- A Scheme frame is a data structure that associates names with values.
- The frame is used to store the arguments and values of procedures.
- New procedures:
  - Built-in primitive procedures
  - Apply (operator, arguments)
  - Evaluating expressions of special forms

### Scheme Pair

- A Scheme pair is a data structure that consists of two elements.
- New procedures:
  - Built-in primitive procedures
  - Apply (operator, arguments)
  - Evaluating expressions of special forms

### Scheme List

- A Scheme list is written as elements in parentheses.
- Each element in a Scheme list can be a combination or atom (primitive).
- A Scheme list can be an expression, a number or a Scheme list.
- New procedures:
  - Built-in primitive procedures
  - Apply (operator, arguments)
  - Evaluating expressions of special forms

### Scheme Interpreter

- The Scheme interpreter takes a sequence of expressions and evaluates them.
- The interpreter uses a stack to store the values of the expressions.
- The interpreter supports recursive procedures.

### Scheme Compiler

- The Scheme compiler translates the abstract syntax tree into machine code.
- New procedures:
  - Built-in primitive procedures
  - Apply (operator, arguments)
  - Evaluating expressions of special forms

### Scheme Execution

- The Scheme interpreter executes the machine code to evaluate the expressions.
- The interpreter supports recursive procedures.
Scheme Contest Winners

Congratulations to Shiyu Li and Henry Maltby for their winning entry in the featherweight division!

And Rain Will Make The Flowers Grow

Bursting into bloom
A bouquet of gracideas
Gift of recursion.

Congratulations to Melanie Cebula for her winning entry in the heavyweight division!

now in 3D (oh god I haven’t slept)

Invest in this code
Get your money back and more
A Pyramid Scheme
Second place in featherweight division: Kevin Lee and Edward Whang.

**finals more like.fml amirite**
Like this arrow, our grades after the finals will point anywhere but up

Second place in heavyweight division: Roger Kuo.

**Majoras Mask**
Little Goblin Here
Now Wearing Majoras Mask
Creating Chaos

Third place in featherweight division: Anand Kuchibotla and Harsha Nukala.

**Amir puts the ”CS” in OlympiCS**
Amir won gold at the Olympics now he is a Kamillionaire

Third place in heavyweight division: Brian Timar.

**Rockets**
rockets rockets rock ets rockets rockets rockets rockets rockets