CS 61A Structure and Interpretation of Computer Programs
Spring 2013

Final

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

Q. 1	Q. 2	Q. 3	Q. 4	Q. 5	Q. 6	Q. 7	Total
/12	/16	/8	/7	/8	/16	/13	/80

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.

```
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.

```
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.

```
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)

        Tree.__init__(self, entry, left, right)

        self.entry = entry

        self.left = left

        self.right = right

        self.depth = depth(self)

        self.is_balanced = is_balanced(self)

        self.depth = depth

        self.is_balanced = is_balanced

    @property

    def depth(self):

        return depth(self)

    @property

    def is_balanced(self):

        return is_balanced(self)
```

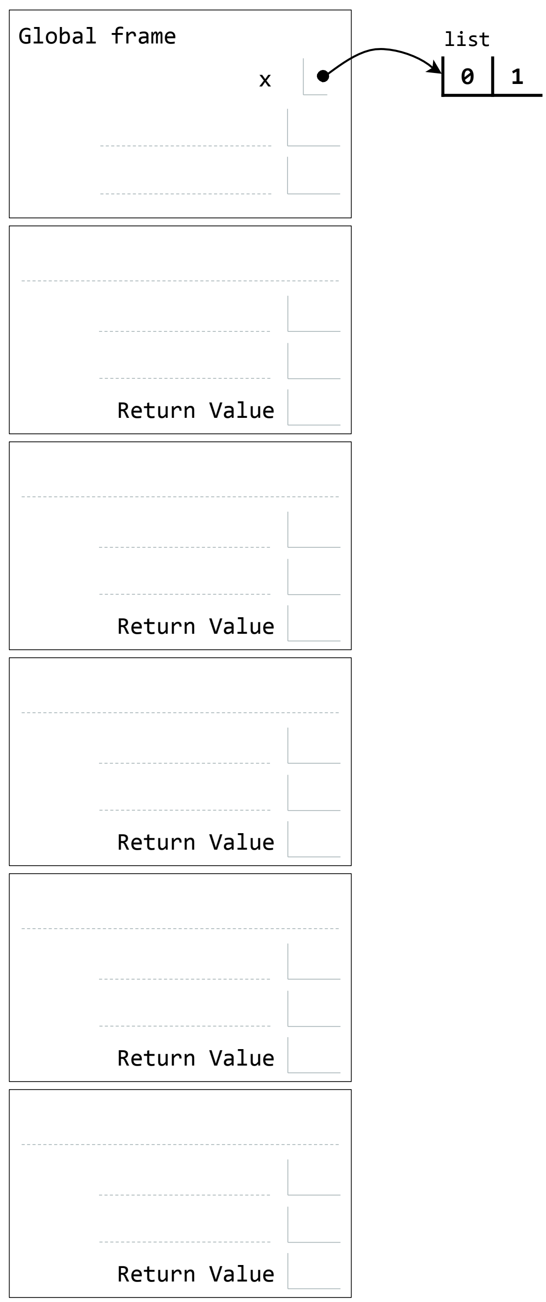
2. (16 points) Binary Tree Huggers are Environmentalists

(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.

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

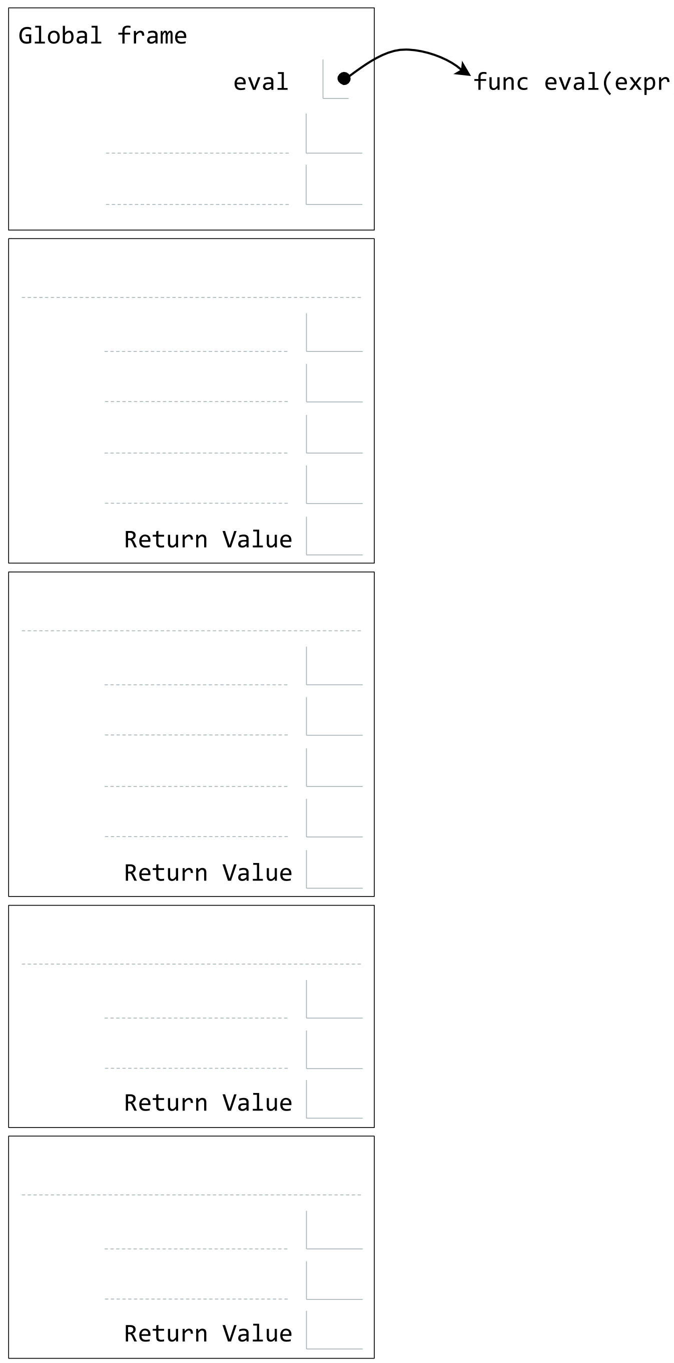


(b) **(9 pt)** Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. You need only show the final state of each frame. *You may not need to use all of the spaces or frames.* You may draw objects that 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.

```
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)
```



3. (8 points) We Recurse at Hard Problems

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

$$
\begin{array}{ccccccccccc}
 & & & & & 1 & & & & & \\
 & & & & 1 & & 1 & & & & \\
 & & & 1 & & 2 & & 1 & & & \\
 & & 1 & & 3 & & 3 & & 1 & & \\
 & 1 & & 4 & & 6 & & 4 & & 1 & \\
1 & & 5 & & 10 & & 10 & & 5 & & 1
\end{array}
$$

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 `pascal` that takes a row index n and an element index k as arguments and computes the kth element in row n, with indexing beginning at 0 for both n and k. *Do **not** compute factorial or any other combinatorial expression as part of your solution.*

```
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 nth row of Pascal's triangle. Your solution should be self-contained; *you may **not** use the* `pascal` *function defined in part (a).*

```
def pascal_gen(n):
    """Return an iterator over all the elements in the nth 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.

```
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.

```
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.

```
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:

```
(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:

Expression	Evaluates to
`(+ 1 4)`	5
`(+ 1 car)`	ERROR
`(cons 1 (cons 2 3))`	
`(cdr '(1 (2) 3))`	
`(z car (list 1 (2) 3))`	
`(z z z)`	
`(z x 3)`	
`(z y 3)`	
`(r x y 3)`	
`(r y x 3)`	

(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.

```
(define (insert lst item index)

  (if _______________________________________________________________

      _______________________________________________________________

      _______________________________________________________________))
```

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

```
;;; 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.

```
(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:

```
(fact (append () ?x ?x))
(fact (append (?a . ?r) ?s (?a . ?t))
      (append ?r ?s ?t))

(fact (foo () () ()))
(fact (foo (?a . ?r) (?b . ?s) ((?a ?b) . ?t))
      (foo ?r ?s ?t))

(fact (bar (?a . ?r) ?s)
      (append ?r (?a) ?s))

(fact (baz ?rel () ()))
(fact (baz ?rel (?a . ?r) (?b . ?s))
      (?rel ?a ?b)
      (baz ?rel ?r ?s))
```

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:

Expression	Interactive Output
`(query (append (1 2) 3 (1 2 3)))`	**Failed.**
`(query (append (1 2) (3 4) ?what))`	**Success!** **what: (1 2 3 4)**
`(query (foo 1 3 (1 3)))`	
`(query (foo (1) (3) (1 3)))`	
`(query (foo (1 2) (3 4) ?what))`	
`(query (bar () ()))`	
`(query (bar (1 2 3 4) ?what))`	
`(query (baz bar ((1 2 3) (4 5) (6)) ?what))`	

(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:

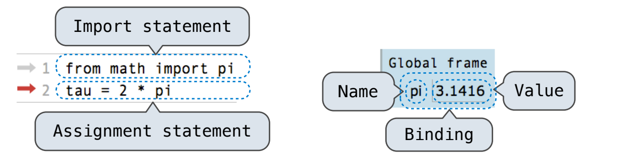
```
logic> (fact (<= a a))
logic> (fact (<= a b))
logic> (fact (<= a c))
logic> (fact (<= b b))
logic> (fact (<= b c))
logic> (fact (<= c c))
logic> (query (sorted ()))
Success!
logic> (query (sorted (a b b c)))
Success!
logic> (query (sorted (b a c)))
Failed.
```

(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:

```
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 ______________________________________________________))
```

(This page intentionally left blank)

(This page intentionally left blank)



Code:

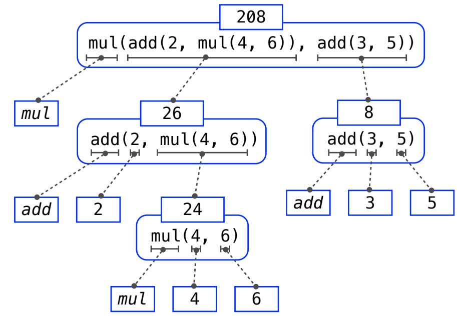
Statements and expressions

Red arrow points to next line.
Gray arrow points to the line just executed

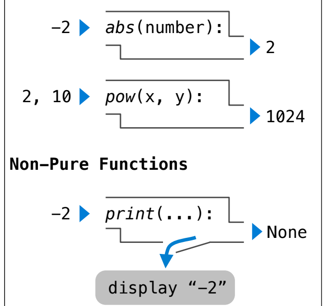
Frames:

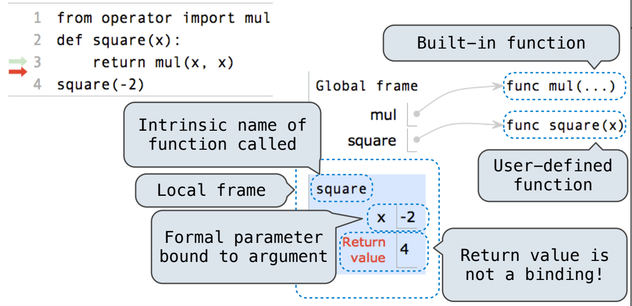
A name is bound to a value

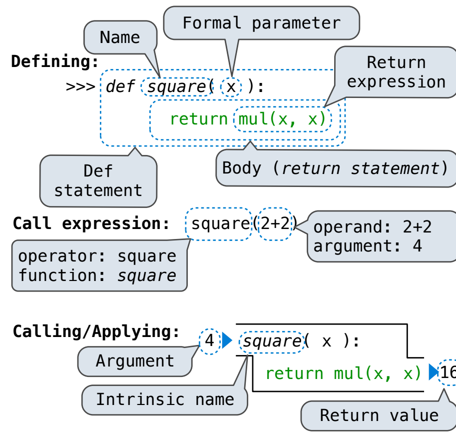
In a frame, there is at most one binding per name

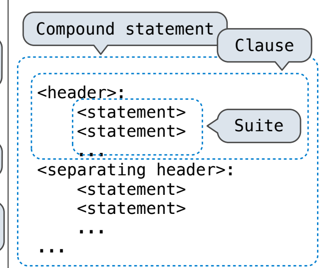


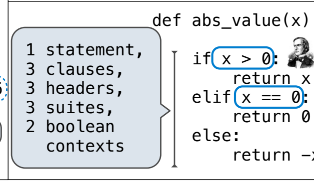
Pure Functions



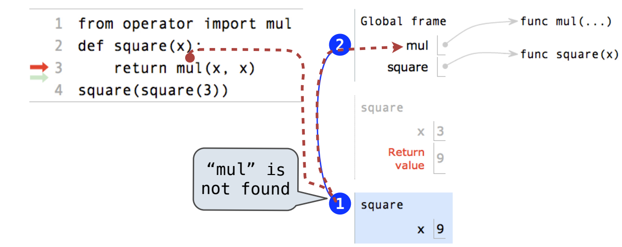


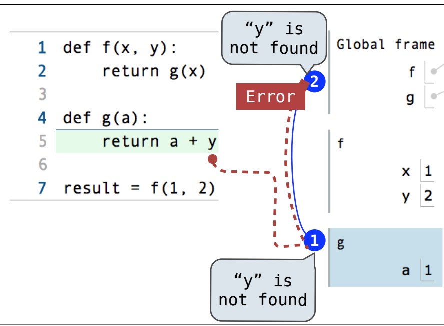






A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.





- An environment is a sequence of frames
- An environment for a non-nested function (no def within def) consists of one local frame, followed by the global frame

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.

Execution rule for conditional statements:

Each clause is considered in order.
1. Evaluate the header's expression.
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:

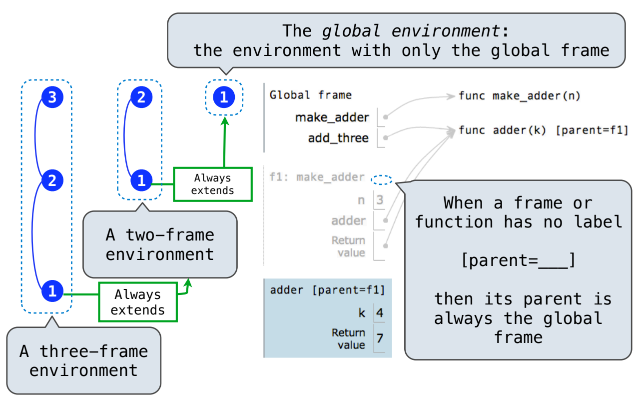
1. Evaluate the subexpression <left>.
2. If the result is a false value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

Evaluation rule for not expressions:

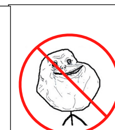
1. Evaluate <exp>; The value is True if the result is a false value, and False otherwise.

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.



A frame *extends* the environment that begins with its parent



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

Nested def statements: Functions defined within other function bodies are bound to names in the local frame

```
def cube(k):
    return pow(k, 3)

def summation(n, term):
    """Sum the first n terms of a sequence.

    >>> summation(5, cube)
    225
    """
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total
```

Function of a single argument (not called term)

A formal parameter that will be bound to a function

The cube function is passed as an argument value

$0 + 1^3 + 2^3 + 3^3 + 4^3 + 5^5$

The function bound to term gets called here

CS 61A Midterm 1 Study Guide — Page 2

```
square = lambda x,y: x * y
```

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

Must be a single expression

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

```
>>> print(4, 2)
4 2
```

```
square = lambda x: x * x
```

VS

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

- Both create a function with the same arguments & behavior
- Both of those functions are associated with the environment in which they are defined
- Both bind that function to the name "square"
- Only the def statement gives the function an intrinsic name

```
def make_adder(n):
    """Return a function that takes one argument k and returns k + n.

    >>> add_three = make_adder(3)
    >>> add_three(4)
    7
    """
    def adder(k):
        return k + n
    return adder
```

A function that returns a function

The name add_three is bound to a function

A local def statement

Can refer to names in the enclosing function

```
make_adder(1)(2)
```

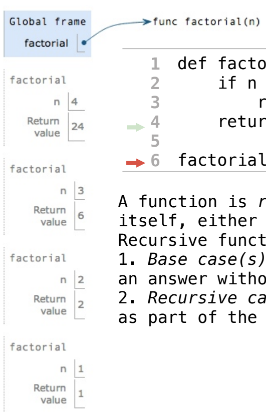
make_adder(1) (2)

Operator: An expression that evaluates to a function value

Operand 0: An expression that evaluates to any value

```
def square(x):
    return mul(x, x)

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

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 mul. **No**



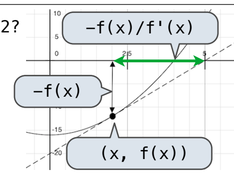

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)

factorial(4)
```

A function is *recursive* if the body calls the function itself, either directly or indirectly

Recursive functions have two important components:
1. *Base case(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

How to find the square root of 2?

```
>>> f = lambda x: x*x - 2
>>> find_zero(f, 1)
1.4142135623730951
```

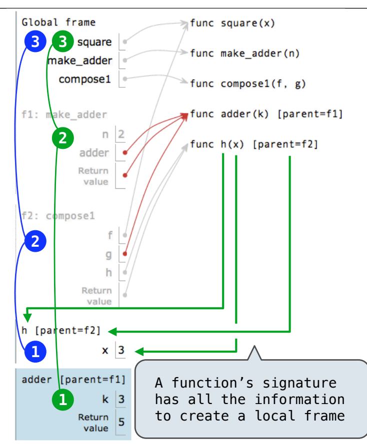


Begin with a function f and an initial guess x

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: $x - \frac{f(x)}{f'(x)}$

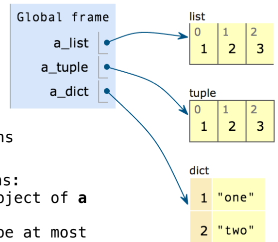
```
def iter_improve(update, done, guess=1, max_updates=1000):
    """Iteratively improve guess with update until done returns a true value.

    >>> iter_improve(golden_update, golden_test)
    1.618033988749895
    """
    k = 0
    while not done(guess) and k < max_updates:
        guess = update(guess)
        k = k + 1
    return guess

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

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

def find_root(f, guess=1):
    """Return a guess of a zero of the function f, near guess.

    >>> from math import sin
    >>> find_root(lambda y: sin(y), 3)
    3.141592653589793
    """
    return iter_improve(newton_update(f), lambda x: f(x) == 0, guess)
```

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

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

def compose1(f, g):
    def h(x):
        return f(g(x))
    return h

compose1(square, make_adder(2))(3)
```



- 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

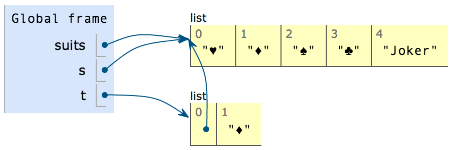
```
a_list = [1, 2, 3]
a_tuple = (1, 2, 3)
a_dict = {1: 'one', 2: 'two'}
```



- 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.

```
suits = ['♥', '♦']
s = suits
t = list(suits)
suits += ['♠', '♣']
t[0] = suits
suits.append('Joker')
```




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

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, ...

range(-2, 2)

```
>>> city = 'Berkeley'
>>> len(city)
8
>>> city[3]
'k'
```

An element of a string is itself a string!

Length. A sequence has a finite length.

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

Generator expressions

(<map exp> for <name> in <iter exp> if <filter exp>)

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

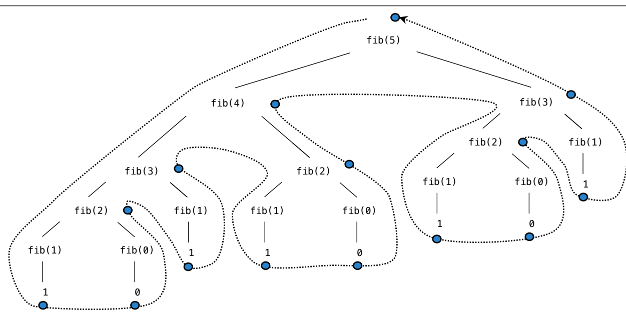
List comprehensions

[<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.

```
>>> suits = ['heart', 'diamond', 'spade', 'club']
>>> from unicodedata import lookup
>>> [lookup('WHITE ' + s.upper() + ' SUIT') for s in suits]
['♡', '♢', '♤', '♧']
```




```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n-1) + fib(n-2)
```

Every object that is an instance of a user-defined class has a unique identity:

```
>>> a = Account('Jim')
>>> b = Account('Jack')
```

Identity testing is performed by "is" and "is not" operators. Binding an object to a new name using assignment **does not** create a new object:

```
>>> a is a
True
>>> a is not b
True
```

```
>>> c = a
>>> c is a
True
```

nonlocal <name>, <name 2>, ...

Effect: Future assignments to that name change its pre-existing binding in the **first non-local frame** of the current environment in which that name is bound.

Python Docs: an "enclosing scope"

From the Python 3 language reference:

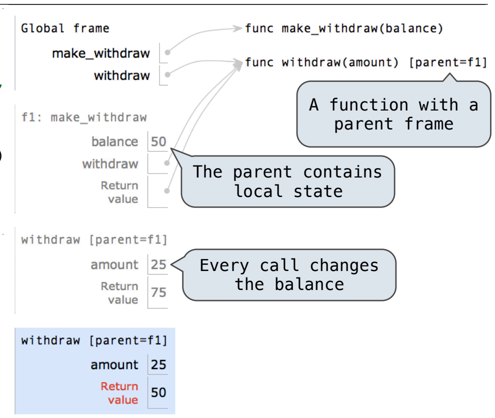
Names listed in a nonlocal statement must refer to pre-existing bindings in an enclosing scope.

Names listed in a nonlocal statement must not collide with pre-existing bindings in the local scope.

x = 2

Status	Effect
•No nonlocal statement •"x" **is not** bound locally	Create a new binding from name "x" to object 2 in the first frame of the current environment.
•No nonlocal statement •"x" **is** bound locally	Re-bind name "x" to object 2 in the first frame of the current env.
•nonlocal x •"x" **is** bound in a non-local frame (but not the global frame)	Re-bind "x" to 2 in the first non-local frame of the current environment in which it is bound.
•nonlocal x •"x" **is not** bound in a non-local frame	SyntaxError: no binding for nonlocal 'x' found
•nonlocal x •"x" **is** bound in a non-local frame •"x" also bound locally	SyntaxError: name 'x' is parameter and nonlocal

```
def make_withdraw(balance):
    def withdraw(amount):
        nonlocal balance
        if amount > balance:
            return 'No funds'
        balance -= amount
    return withdraw

withdraw = make_withdraw(100)
withdraw(25)
withdraw(25)
```



Python pre-computes which frame contains each name before executing the body of a function.

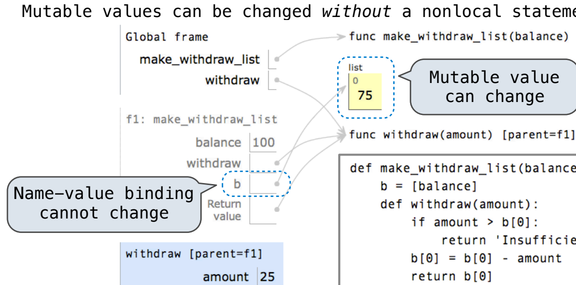
Therefore, within the body of a function, all instances of a name must refer to the same frame.

```
def make_withdraw(balance):
    def withdraw(amount):
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount
        return balance
    return withdraw

wd = make_withdraw(20)
wd(5)
```

Local assignment

UnboundLocalError: local variable 'balance' referenced before assignment

Mutable values can be changed *without* a nonlocal statement.




```
def make_withdraw_list(balance):
    b = [balance]
    def withdraw(amount):
        if amount > b[0]:
            return 'Insufficient funds'
        b[0] = b[0] - amount
        return b[0]
    return withdraw

withdraw = make_withdraw_list(100)
withdraw(25)
```

```
def pig_latin(w):
    if starts_with_a_vowel(w):
        return w + 'ay'
    return pig_latin(w[1:] + w[0])

def starts_with_a_vowel(w):
    return w[0].lower() in 'aeiou'
```

- The **def statement header** is similar to other functions
- Conditional statements check for **base cases**
- Base cases are evaluated **without recursive calls**
- Typically, all other cases are evaluated **with recursive calls**

```
class <name>(<base class>):
    <suite>
```

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

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

Class attribute

Constructor

Methods

Assignment statements with a dot expression on their left-hand side affect attributes 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

```
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
>>> tom_account.interest
0.02
>>> jim_account.interest
0.02
>>> tom_account.interest
0.02
>>> Account.interest = 0.04
>>> tom_account.interest
0.04
```

```
>>> jim_account.interest = 0.8
>>> jim_account.interest
0.8
>>> tom_account.interest
0.04
>>> Account.interest = 0.05
>>> tom_account.interest
0.05
>>> jim_account.interest
0.8
```

Instance Attribute Assignment : tom_account.interest = 0.08

This expression evaluates to an object

But the name ("interest") is **not** looked up

Attribute assignment statement adds or modifies the "interest" attribute of tom_account

```
class CheckingAccount(Account):
    withdraw_fee = 1
    interest = 0.01
    def withdraw(self, amount):
        return Account.withdraw(self, amount + self.withdraw_fee)
```

Base class

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.

```
>>> ch = CheckingAccount('T')
>>> ch.interest
0.01
>>> ch.deposit(20)
20
>>> ch.withdraw(5)
14
```

```
class SavingsAccount(Account):
    deposit_fee = 2
    def deposit(self, amount):
        return Account.deposit(self, amount - self.deposit_fee)

class AsSeenOnTVAccount(CheckingAccount, SavingsAccount):
    def __init__(self, account_holder):
        self.holder = account_holder
        self.balance = 1               # A free dollar!
```

```
>>> a = Account('Jim')
```

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.

```
class Account(object):
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
```

A mutable Rlist implementation using message passing

```
def mutable_rlist():
    contents = empty_rlist
    def dispatch(message, value=None):
        nonlocal 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)
        elif message == 'pop_first':
            f = first(contents)
            contents = rest(contents)
            return f
        elif message == 'str':
            return str(contents)
    return dispatch
```

A bank account implemented using dispatch dictionaries

```
def account(balance):
    def withdraw(amount):
        if amount > dispatch['balance']:
            return 'Insufficient funds'
        dispatch['balance'] -= amount
        return dispatch['balance']
    def deposit(amount):
        dispatch['balance'] += amount
        return dispatch['balance']
    dispatch = {'balance': balance, 'withdraw': withdraw,
                'deposit': deposit}
    return dispatch
```

A simple container implemented using two accessor methods

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

```
class ComplexRI(object):
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag
    @property
    def magnitude(self):
        return (self.real ** 2 + self.imag ** 2) ** 0.5
```

Special decorator: "Call this function on attribute look-up"

Type dispatching: Define a different function for each possible combination of types for which an operation is valid

```
def iscomplex(z):
    return type(z) in (ComplexRI, ComplexMA)

def isrational(z):
    return type(z) == Rational

def add_complex_and_rational(z, r):
    return ComplexRI(z.real + r.numer/r.denom, z.imag)

def add_by_type_dispatching(z1, z2):
    """Add z1 and z2, which may be complex or rational."""
    if iscomplex(z1) and iscomplex(z2):
        return add_complex(z1, z2)
    elif iscomplex(z1) and isrational(z2):
        return add_complex_and_rational(z1, z2)
    elif isrational(z1) and iscomplex(z2):
        return add_complex_and_rational(z2, z1)
    else:
        add_rational(z1, z2)
```

Converted to a real number (float)

```
class Rlist(object):

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

    empty = EmptyList()

    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest

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

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

The base case

A recursive call

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

def map_rlist(s, fn):
    if s is Rlist.empty:
        return s
    rest = map_rlist(s.rest, fn)
    return Rlist(fn(s.first),rest)

def count_entries(tree):
    if tree is None:
        return 0
    left = count_entries(tree.left)
    right = count_entries(tree.right)
    return 1 + left + right
```

A valid tree has no cycles!

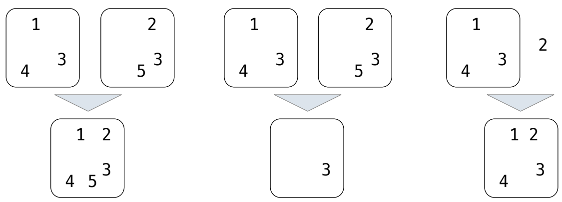
Valid if these are None or a Tree instance

1. Attempt to coerce arguments into values of the same type
2. Apply type-specific (not cross-type) operations

```
def coerce_apply(operator_name, x, y):
    tx, ty = type_tag(x), type_tag(y)
    if tx != ty:
        if (tx, ty) in coercions:
            tx, x = ty, coercions[(tx, ty)](x)
        elif (ty, tx) in coercions:
            ty, y = tx, coercions[(ty, tx)](y)
        else:
            return 'No coercion possible.'
    key = (operator_name, tx)
    return coerce_apply.implementations[key](x, y)
```

CS 61A Final Exam Study Guide – Page 1

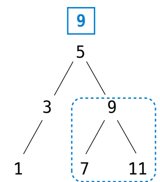
The interface for *sets*:
- Membership testing: Is a value an element of a set?
- Adjunction: Return a set with all elements in s and a value v.
- Union: Return a set with all elements in set1 **or** set2.
- Intersection: Return a set with any elements in set1 **and** set2.



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

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

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



Proposal	1	2	3
Membership	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$
Adjunction	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$
Union	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n)$
Intersection	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n)$

Exceptions are raised with a raise statement.

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

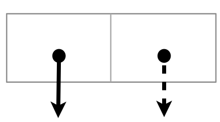
The <try suite> is executed first;

If, during the course of executing the <try suite>, an exception is raised that is not handled otherwise, and

If the class of the exception inherits from <exception class>, then

The <except suite> is executed, with <name> bound to the exception

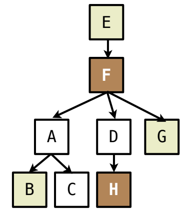
Streams are lazily computed recursive lists




```
class Stream(Rlist):
    def __init__(self, first, compute_rest=lambda: Stream.empty):
        if not callable(compute_rest):
            raise TypeError('compute_rest must be callable')
        self.first = first
        self._compute_rest = compute_rest
        self._rest = None

    @property
    def rest(self):
        if self._compute_rest is not None:
            self._rest = self._compute_rest()
            self._compute_rest = None
        return self._rest

    def __len__(self):
        raise NotImplementedError('length not supported on Streams’)
    def __repr__(self):
        return 'Stream({0}, <...>)'.format(repr(self.first))

def integer_stream(first=1):
    def compute_rest():
        return integer_stream(first+1)
    return Stream(first, compute_rest)

def filter_stream(fn, s):
    if s is Stream.empty:
        return s
    def compute_rest():
        return filter_stream(fn, s.rest)
    if fn(s.first):
        return Stream(s.first, compute_rest)
    else:
        return compute_rest()

def map_stream(fn, s):
    if s is Stream.empty:
        return s
    def compute_rest():
        return map_stream(fn, s.rest)
    return Stream(fn(s.first),
                  compute_rest)

def primes(pos_stream):
        def not_divisible(x):
            return x % pos_stream.first != 0
        def compute_rest():
            return primes(filter_stream(not_divisible, pos_stream.rest))
        return Stream(pos_stream.first, 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.




```
logic> (fact (parent delano herbert))
logic> (fact (parent abraham barack))
logic> (fact (parent abraham clinton))
logic> (fact (parent fillmore abraham))
logic> (fact (parent fillmore delano))
logic> (fact (parent fillmore grover))
logic> (fact (parent eisenhower fillmore))
```

Relations can contain relations in addition to atoms.

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

Variables can refer to atoms or relations in queries.

```
logic> (query (parent abraham ?child))
Success!
child: barack
child: clinton

logic> (query (dog (name clinton) (color ?color)))
Success!
color: white

logic> (query (dog (name clinton) ?info))
Success!
info: (color white)
```

A fact can include multiple relations and variables as well:

(fact <conclusion> <hypothesis$_0$> <hypothesis$_1$> ... <hypothesis$_N$>)

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

```
logic> (fact (child ?c ?p) (parent ?p ?c))

logic> (query (child herbert delano))
Success!

logic> (query (child eisenhower clinton))
Failure.

logic> (query (child ?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.

```
logic> (fact (ancestor ?a ?y) (parent ?a ?y))
logic> (fact (ancestor ?a ?y) (parent ?a ?z) (ancestor ?z ?y))

logic> (query (ancestor ?a herbert))
Success!
a: delano
a: fillmore
a: eisenhower
```

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

```
(parent delano herbert)      ; (1), a simple fact
(ancestor delano herbert)    ; (2), from (1) and the 1st ancestor fact
(parent fillmore delano)     ; (3), a simple fact
(ancestor fillmore herbert)  ; (4), from (2), (3), & the 2nd ancestor fact
```

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

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

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

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

class LetterIterable(object):
    """An iterable over letters."""
    def __iter__(self):
        current = 'a'
        while current <= 'd':
            yield current
            current = chr(ord(current)+1)
```

- A generator is an iterator backed by a generator function.
- When a generator function is called, it returns a generator.

```
>>> letters = Letters()
>>> letters.__next__()
'a'
>>> letters.__next__()
'b'
>>> letters.__next__()
'c'
>>> letters.__next__()
'd'
>>> letters.__next__()
Traceback ...
StopIteration
>>> for x in Letters():
        print(x)
a
b
c
d
>>> for x in letters_generator():
        print(x)
a
b
c
d
>>> for x in LetterIterable():
        print(x)
a
b
c
d
```

CS 61A Final Exam Study Guide – Page 2

Scheme programs consist of expressions, which can be:
- Primitive expressions: 2, 3.3, true, +, quotient, ...
- Combinations: (quotient 10 2), (not true), ...

Numbers are self-evaluating; symbols are bound to values.
Call expressions have an operator and 0 or more operands.

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

```
> (define pi 3.14)
> (* pi 2)
6.28
```

```
> (define (abs x)
    (if (< x 0)
        (- x)
        x))
> (abs -3)
3
```

Lambda expressions evaluate to anonymous functions.

```
(lambda (<formal-parameters>) <body>)
```

λ

Two equivalent expressions:

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

An operator can be a call expression too:

```
((lambda (x y z) (+ x y (square z))) 1 2 3)
```

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 (cons 4 nil))))
(1 2 3 4)
```

Not a well-formed list!

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

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

No sign of "a" and "b" in the resulting value

Quotation is used to refer to symbols directly in Lisp.

```
> (list 'a 'b)
(a b)
> (list 'a b)
(a 2)
```

Symbols are now values

Quotation can also be applied to combinations to form lists.

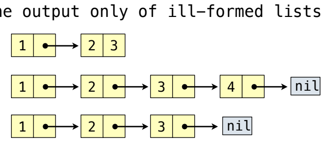
```
> (car '(a b c))
a
> (cdr '(a b c))
(b c)
```

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

```
> (cdr (cdr '(1 2 . 3)))
3
```

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

```
> '(1 2 . 3)
(1 2 . 3)
> '(1 2 . (3 4))
(1 2 3 4)
> '(1 2 3 . nil)
(1 2 3)
> (cdr '((1 2) . (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 (x y) (f (+ x x))))
> (g 3 7)
13
```

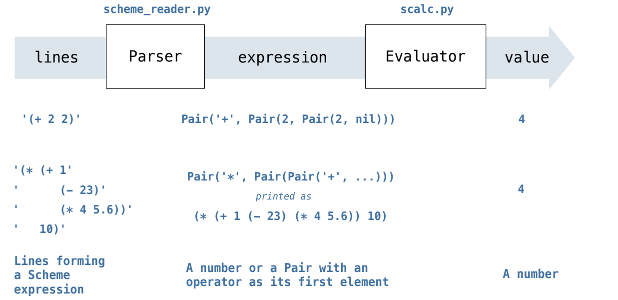
A *race condition* is when multiple threads concurrently access the same data, and at least one mutates it

```
counter = [0]
def increment():
    counter[0] = counter[0] + 1
```

Thread 0	Thread 1
read counter[0]: 0	
	read counter[0]: 0
calculate 0 + 1: 1	
write 1 -> counter[0]	
	calculate 0 + 1: 1
	write 1 -> counter[0]

Possible final values for counter[0]: 1, 2

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



A Scheme list is written as elements in parentheses:

($\text{<element}_0\text{>}$ $\text{<element}_1\text{>}$ ... $\text{<element}_n\text{>}$)

A recursive Scheme list

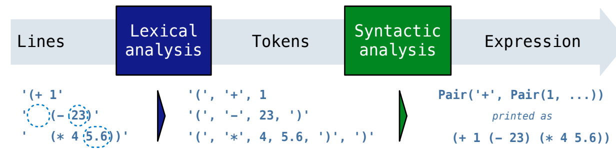
Each <element> can be a combination or atom (primitive).

(+ (* 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.



- Iterative process
- Checks for malformed tokens
- Determines types of tokens
- Processes one line at a time

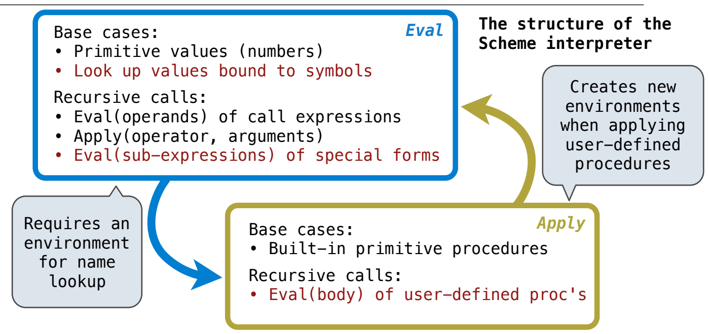
- Tree-recursive process
- Balances parentheses
- Returns tree structure
- Processes multiple lines

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.

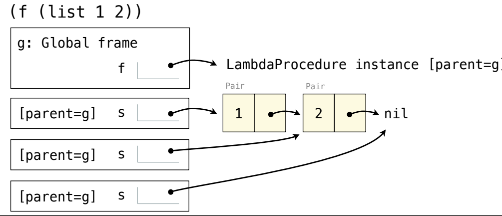
Base case: symbols and numbers
Recursive call: scheme_read sub-expressions and combine them

The structure of the Scheme interpreter



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.

```
(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 (factorial n k)
  (if (= n 0) k
    (factorial (- n 1)
               (* k n))))
```

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

Not a tail call

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

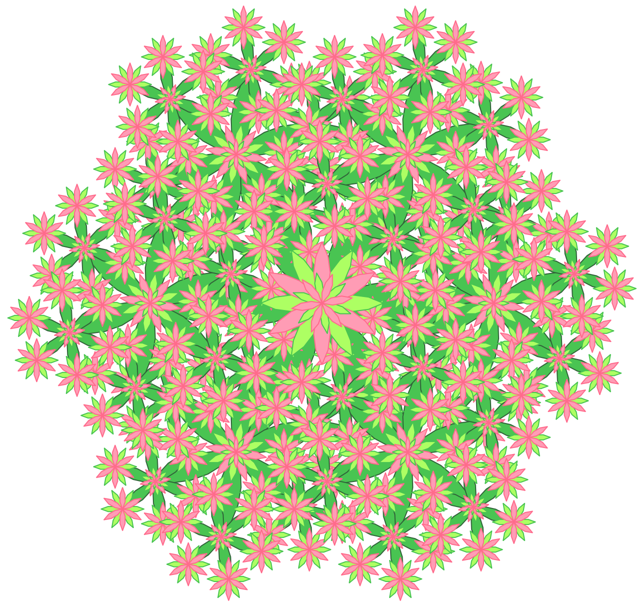
Recursive call is a tail call

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

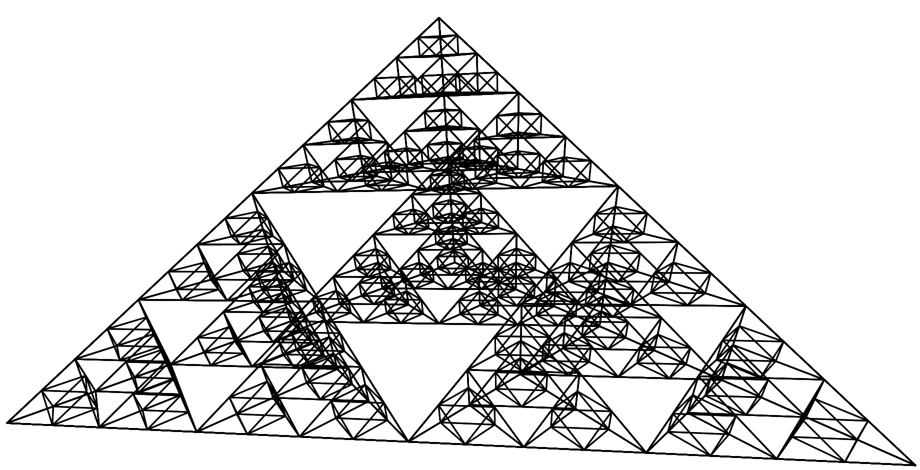
Bursting into bloom
A bouquet of gracideas
Gift of recursion.



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

now in 3D (oh god I haven't slept)

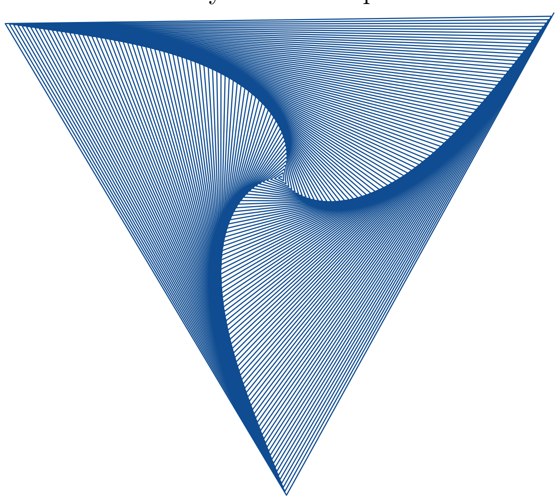
Invest in this code
Get your money back and more
A Pyramid Scheme



Second place in featherweight division: Kevin Lee and Edward Whang.

finals more like fml amirite

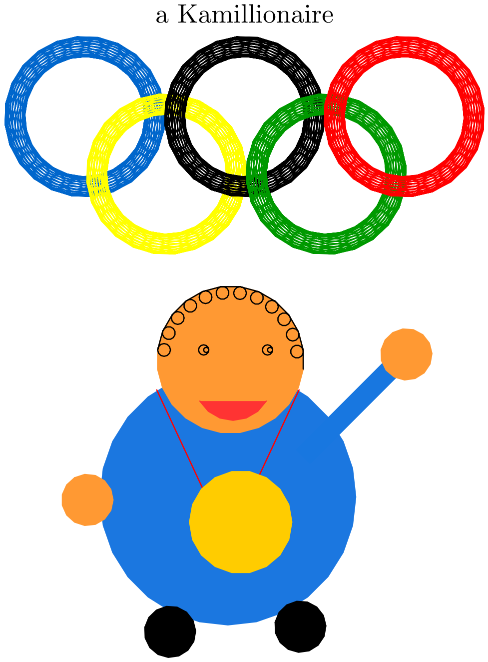
Like this arrow, our
grades after the finals will point
anywhere but up



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

Amir puts the "CS" in OlympiCS

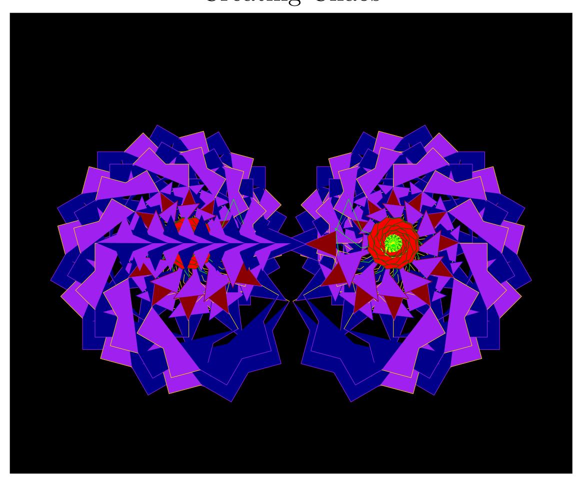
Amir won gold at
the Olympics now he is
a Kamillionaire



Second place in heavyweight division: Roger Kuo.

Majoras Mask

Little Goblin Here
Now Wearing Majoras Mask
Creating Chaos



Third place in heavyweight division: Brian Timar.

Rockets

rockets rockets rock
ets rockets rockets rockets
rockets rockets rock