CS 61A Structure and Interpretation of Computer Programs

Fall 2012

Final Examination

INSTRUCTIONS

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

Last name	
First name	
SID	
Login	
TA & section time	
Name of the person to your left	
Name of the person to your right	
All the work on this exam is my own. **(please sign)**	

For staff use only

Q. 1	Q. 2	Q. 3	Q. 4	Total
/20	/16	/30	/14	/80

THIS PAGE INTENTIONALLY LEFT BLANK

1. **(20 points) Bank Rewrite**

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

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

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

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

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

Expression	Evaluates to
5*5	25
feb[jan[0]]	
mar(jan, feb)	
jan	
next(iter(jan))	
len(mar(range(5, 50), range(20, 200)))	

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

Assume that you have started Python 3 and executed the following statements **after executing the Stream class statement from the Final Exam Study Guide**:

```
from operator import add, mul

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

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

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

Expression	Evaluates to
`(1, rock.first)`	(1, 3)
`(rock.rest.first, rock.rest.rest.first)`	
`(fire.first, fire.rest.first)`	
`fire.rest is fire.rest`	
`(ash.first, ash.rest.first)`	
`ash.rest.rest.rest.first`	

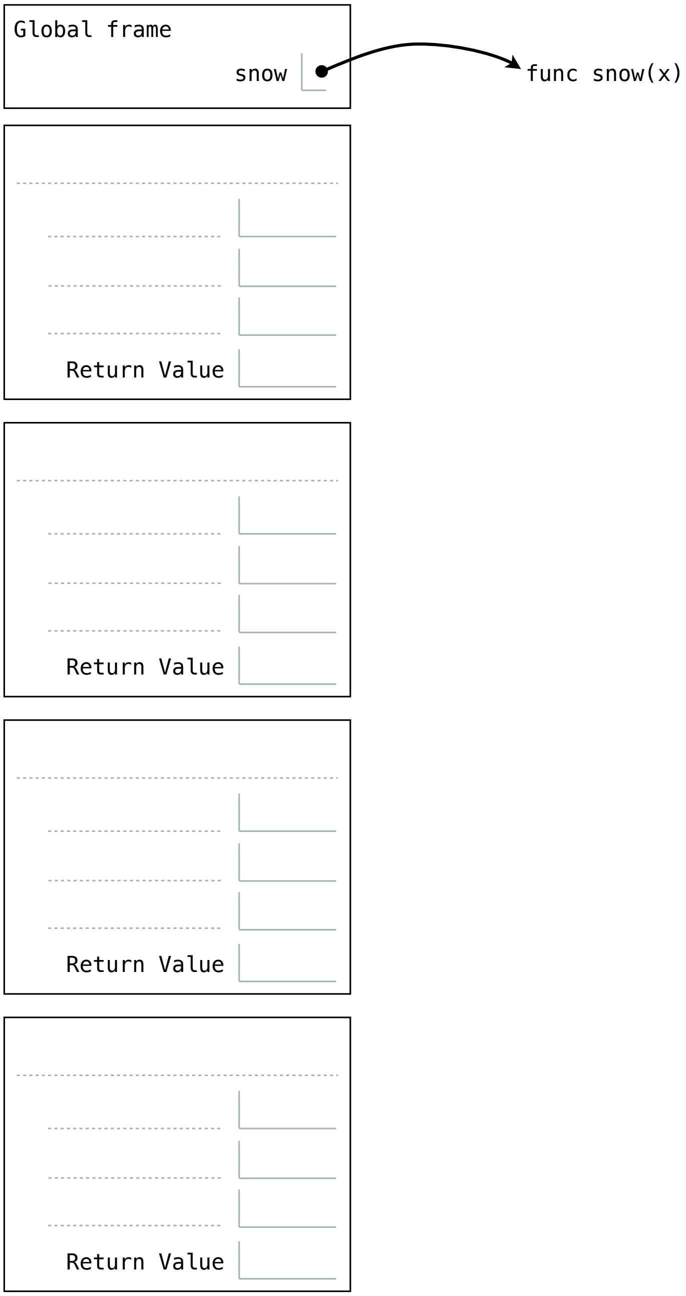
2. (16 points) Web Rater Ink

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

A complete answer will:

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

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

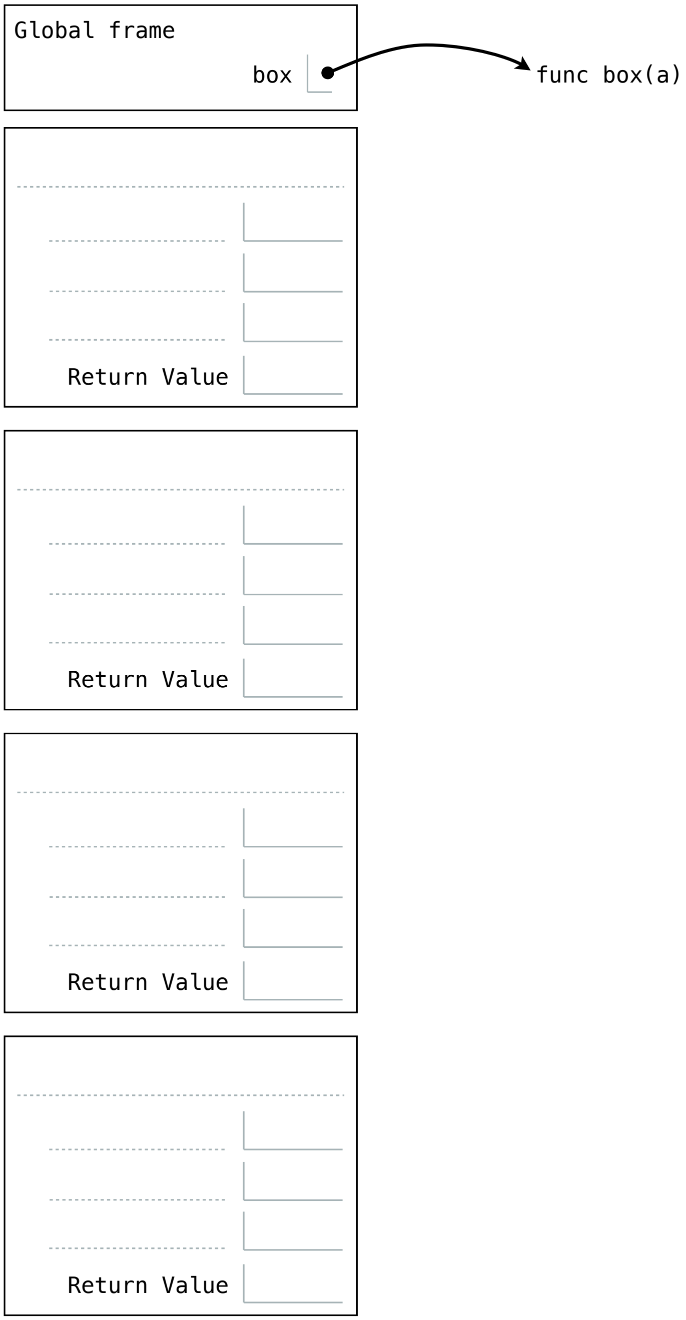


(b) **(8 pt)** Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. *You may not need to use all of the spaces or frames.*

A complete answer will:

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

```
def box(a):
    def box(b):
        def box(c):
            nonlocal a
            a = a + c
            return (a, b)
        return box
    gift = box(1)
    return (gift(2), gift(3))
box(4)
```



3. (30 points) Twin Breaker

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

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

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

        last, count = None, 0

        self.runs = []

        for elem in elements:

            if ______________________________________________________:

                self.runs.append(________________________________________)

            if ______________________________________________________:

                last, count = ________________________________________

            else:

                count += 1

        ______________________________________________________________

    def __len__(self):

        return sum(______________________________________________________)

    def __getitem__(self, k):

        run = 0

        while _______________________________________________________:

            k, run = __________________________________________________

        return self.runs[run][1]
```

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

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

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

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

    if tree is None:

        return 0

    return 1 + max(__________________________________________________

                   __________________________________________________

                   __________________________________________________)

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

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

    if tree is None:

        return 0

    return max(______________________________________________________

               ______________________________________________________

               ______________________________________________________

               ______________________________________________________)
```

(c) (8 pt) Given a set of unique positive integers `s` and a maximum sum `m`, the `pack` function returns a subset of `s` with the largest sum less than or equal to `m`. Fill in the blanks below, so that all doctests pass. *Assume that sets are printed in sorted order, regardless of how they are constructed.*

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

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

    if len(s) == 0:

        return set()

    if s[0] > m:


        return __________________________________________________________


    with_s0 = {s[0]}.union(______________________________________________)


    without = ____________________________________________________________


    if ___________________________________________________________________:

        return with_s0

    else:

        return without
```

(d) **(8 pt)** Cross out lines from the implementation of the `IterableTree` class below so that all doctests pass **and the implementation contains as few lines of code as possible**. Don't cross out any docstrings or doctests.

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

```
class IterableTree(object):

class IterableTree(Tree):

    def __init__(self, entry, left=None, right=None):

        Tree.__init__(entry, left, right)

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

    def __iter__(self):
        """Yield the entries of this tree.

        >>> T = IterableTree
        >>> t = T('A', T(2, T('C'), T(4)), T('E', None, T(6)))
        >>> list(t)
        ['A', 2, 'C', 4, 'E', 6]
        """

        yield self.entry

        yield entry

        for branch in (self.left, self.right):

            if branch:

            if self.branch:

                branch = iter(branch)

                for entry in branch:

                for entry in branch():

                    yield self.entry

                    yield entry

        yield self.entry

        yield entry
```

4. (14 points) Winter Break

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

```
(define j

  (mu (c k)

    (if (< c n)

        (j (+ c 2) (- k 1))

        k)))

(define (f n)

  (define c n)

  (j 0 0))
```

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

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

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

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

Calls to scheme_apply (circle one): 1 2 3 4

(d) **(4 pt)** Fill in two facts below to complete the definitions of the relations `reversed` and `palindrome`. The `reversed` relation indicates that the first list contains the same elements as the second, but in reversed order. The `palindrome` relation indicates that a list is the same backward and forward.

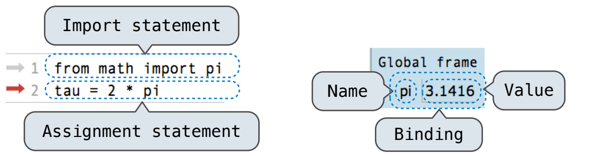
```
logic> (fact (append-to-form () ?x ?x))

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

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

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


             ____________________________________________________________)

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

logic> (fact (palindrome ?s)


             ____________________________________________________________)

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

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

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

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

$f(n) =$

CS 61A Midterm 1 Study Guide – Page 1



```
from math import pi
tau = 2 * pi
```

Code (left):

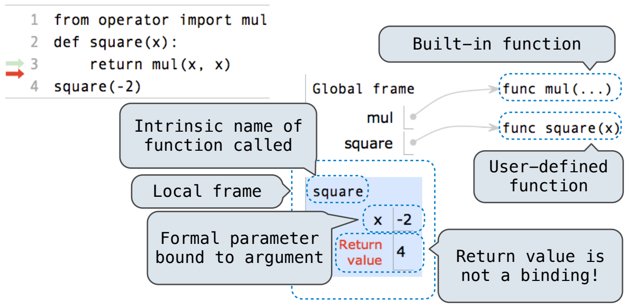
Statements and expressions

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

Frames (right):

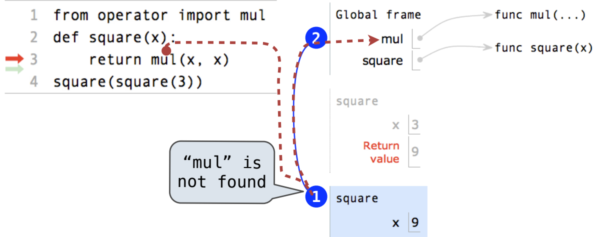
A name is bound to a value

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



```
from operator import mul
def square(x):
    return mul(x, x)
square(-2)
```

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



```
from operator import mul
def square(x):
    return mul(x, x)
square(square(3))
```

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:

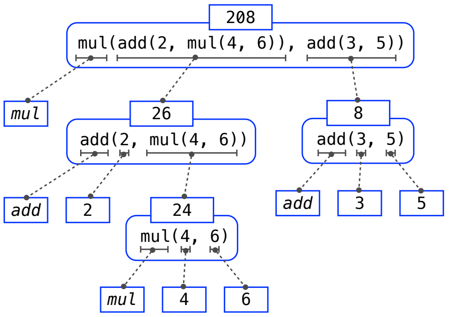
1. Evaluate the subexpression <left>.
2. If the result is a false value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

Evaluation rule for not expressions:

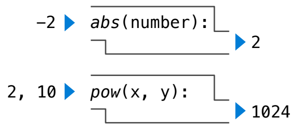
1. Evaluate <exp>; The value is True if the result is a false value, and False otherwise.

Execution rule for while statements:

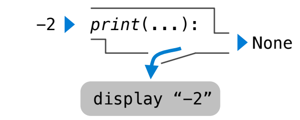
1. Evaluate the header’s expression.
2. If it is a true value, execute the (*whole*) suite, then return to step 1.

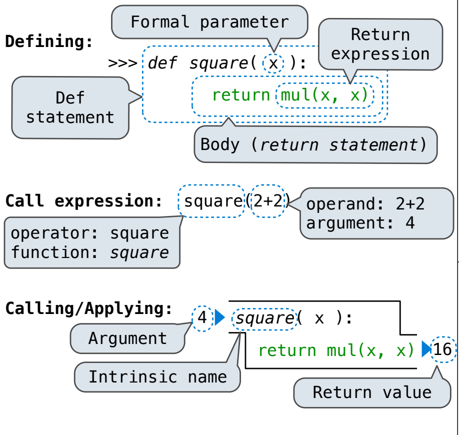


Pure Functions



Non-Pure Functions





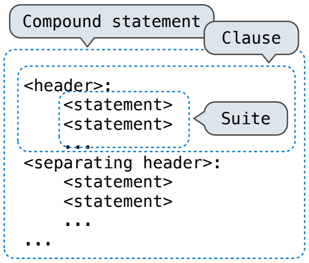
Defining:

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

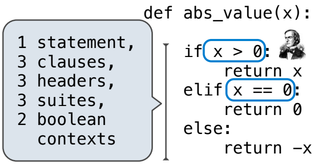
Call expression: square(2+2)

Calling/Applying:

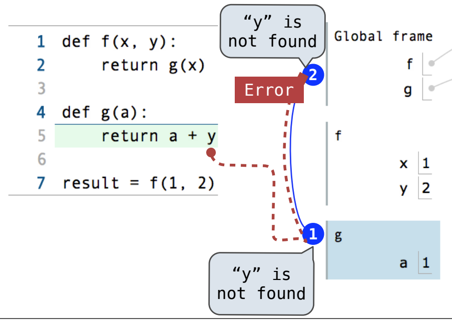
```
square( x ):
    return mul(x, x)
```



```
<header>:
    <statement>
    <statement>
    ...
<separating header>:
    <statement>
    <statement>
    ...
...
```

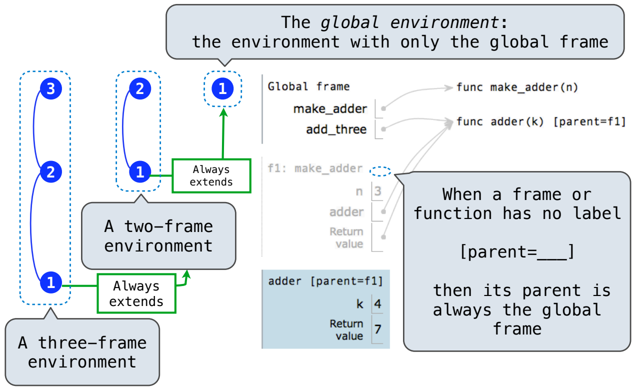


```
def abs_value(x):
    if x > 0:
        return x
    elif x == 0:
        return 0
    else:
        return -x
```



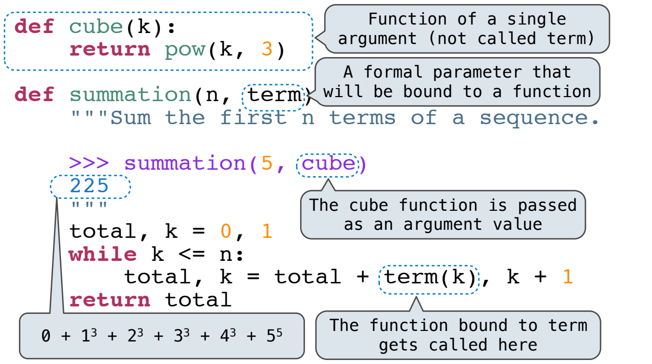
```
def f(x, y):
    return g(x)

def g(a):
    return a + y

result = f(1, 2)
```

- 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



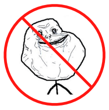
The global environment: the environment with only the global frame
3
2
1
2
1
1
Always extends
A two-frame environment
Always extends
A three-frame environment
Global frame
make_adder
add_three
func make_adder(n)
func adder(k) [parent=f1]
f1: make_adder
n 3
adder
Return value
adder [parent=f1]
k 4
Return value 7
When a frame or function has no label [parent=___] then its parent is always the global frame

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



```
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 + 1³ + 2³ + 3³ + 4³ + 5⁵
The function bound to term gets called here



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

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

```
@trace1
def triple(x):
    return 3 * x
```

is identical to

```
def triple(x):
    return 3 * x
triple = trace1(triple)
```

```
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
```

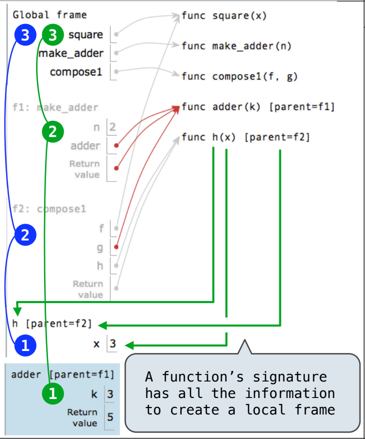
A function that returns a function

The name add_three is bound to a function

A local def statement

Can refer to names in the enclosing function

```
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)
```



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

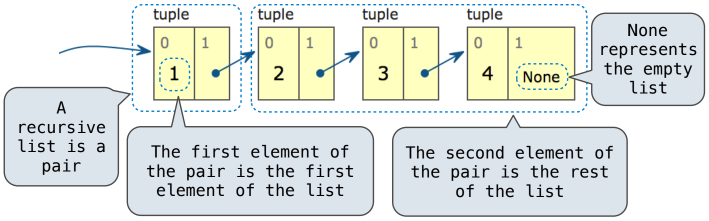
- Every user-defined function has a parent frame
- The parent of a function is the frame in which it was defined
- Every local frame has a parent frame
- The parent of a frame is the parent of the function called

- Compound objects combine objects together
- An *abstract data type* lets us manipulate compound objects as units
- Programs that use data isolate two aspects of programming:
  - How data are represented (as parts)
  - How data are manipulated (as units)
- Data abstraction: A methodology by which functions enforce an abstraction barrier between ***representation*** and ***use***

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



tuple
tuple
tuple
tuple
0
1
1
0
1
2
0
1
3
0
1
4
None
A recursive list is a pair
The first element of the pair is the first element of the list
The second element of the pair is the rest of the list
None represents the empty list

```
empty_rlist = None
def rlist(first, rest):
    """Make a recursive list from its first element and the rest."""
    return (first, rest)
def first(s):
    """Return the first element of a recursive list s."""
    return s[0]
def rest(s):
    """Return the rest of the elements of a recursive list s."""
    return s[1]
```

If a recursive list s is constructed from a first element f and a recursive list r, then
- first(s) returns f, and
- rest(s) returns r, which is a recursive list.

```
def len_rlist(s):
    """Return the length of recursive list s."""
    length = 0
    while s != empty_rlist:
        s, length = rest(s), length + 1
    return length

def getitem_rlist(s, i):
    """Return the element at index i of rlist s."""
    while i > 0:
        s, i = rest(s), i - 1
    return first(s)
```

Length. A sequence has a finite length.

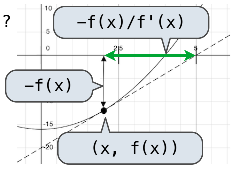
Element selection. A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

```
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

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 mul_rational(x, y):
    return rational(numer(x) * numer(y), denom(x) * denom(y))
```

Constructor

Selectors

```
def add_rational(x, y):
    nx, dx = numer(x), denom(x)
    ny, dy = numer(y), denom(y)
    return rational(nx * dy + ny * dx, dx * dy)
def eq_rational(x, y):
    return numer(x) * denom(y) == numer(y) * denom(x)
```

```
def rational(n, d):
    """Construct a rational number x that represents n/d."""
    return (n, d)

from operator import getitem
def numer(x):
    """Return the numerator of rational number x."""
    return getitem(x, 0)

def denom(x):
    """Return the denominator of rational number x."""
    return getitem(x, 1)
```

```
def pair(x, y):
    """Return a functional pair."""
    def dispatch(m):
        if m == 0:
            return x
        elif m == 1:
            return y
    return dispatch
def getitem_pair(p, i):
    """Return the element at index i of pair p."""
    return p(i)
```

This function represents a pair

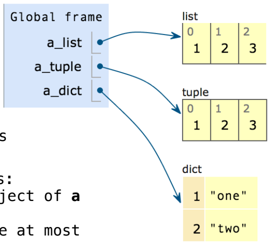
```
from operator import floordiv, mod
def divide_exact(n, d):
    """Return the quotient and remainder of dividing N by D.

    >>> q, r = divide_exact(2012, 10)
    >>> q
    201
    >>> r
    2
    """
    return floordiv(n, d), mod(n, d)
```

Multiple assignment to two names

Multiple return values, separated by commas

CS 61A Midterm 2 Study Guide – Page 1

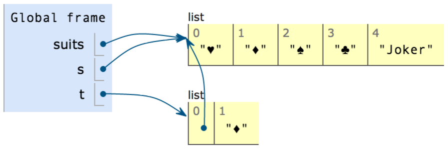
```
a_list = [1, 2, 3]
a_tuple = (1, 2, 3)
a_dict = {1: 'one', 2: 'two'}
```



- 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.

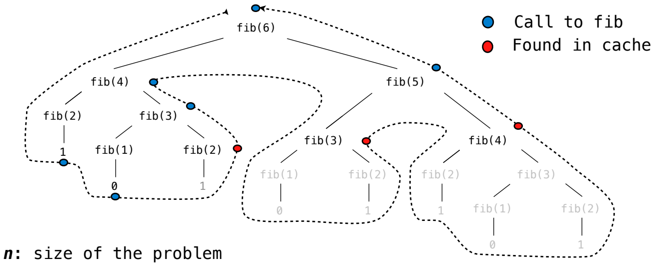
List comprehensions

[<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]
['♡', '♢', '♤', '♧']
```



n: size of the problem
***R*(*n*)**: Measurement of some resource used (time or space)

$$R(n) = \Theta(f(n))$$

means that there are constants k_1 and k_2 such that

$$k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n)$$

for sufficiently large values of ***n***.

$\Theta(b^n)$ $\cdots$ $\Theta(n^3)$ $\Theta(n^2)$ $\Theta(n)$ $\Theta(\log n)$ $\Theta(1)$

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:

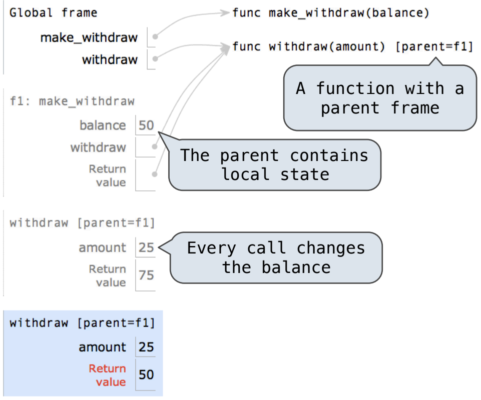
Names listed in a nonlocal statement must refer to pre-existing bindings in an enclosing scope.

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 balance
    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.

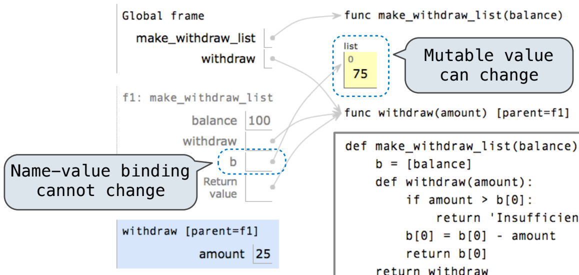
Therefore, within the body of a function, all instances of a name must refer to the same frame.

```
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**

CS 61A Midterm 2 Study Guide — Page 2

```
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
```

```
def make_instance(cls):
    def get_value(name):
        if name in attributes:
            return attributes[name]
        else:
            value = cls['get'](name)
            return bind_method(value, instance)
    def set_value(name, value):
        attributes[name] = value
    attributes = {}
    instance = {'get': get_value, 'set': set_value}
    return instance
def bind_method(value, instance):
    if callable(value):
        def method(*args):
            return value(instance, *args)
        return method
    else:
        return value
def make_class(attributes={}, base_class=None):
    def get_value(name):
        if name in attributes:
            return attributes[name]
        elif base_class is not None:
            return base_class['get'](name)
    def set_value(name, value):
        attributes[name] = value
    def new(*args):
        return init_instance(cls, *args)
    cls = {'get': get_value, 'set': set_value, 'new': new}
    return cls
def init_instance(cls, *args):
    instance = make_instance(cls)
    init = cls['get']('__init__')
    if init is not None:
        init(instance, *args)
    return instance
def make_account_class():
    interest = 0.02
    def __init__(self, account_holder):
        self['set']('holder', account_holder)
        self['set']('balance', 0)
    def deposit(self, amount):
        new_balance = self['get']('balance') + amount
        self['set']('balance', new_balance)
        return self['get']('balance')
    ...
    return make_class(locals())
Account = make_account_class()
```

The class of the instance

Match name against instance attributes

Look up the name in the class

Assignment affects instance attributes

Class attribute lookup

Common dispatch dictionary pattern

Dispatch dictionary

Special constructor name is fixed here

```
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)

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)
```

```
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

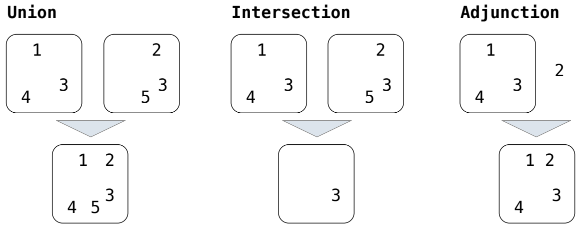
A recursive call

```
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_leaves(tree):
    if type(tree) != tuple:
        return 1
    return sum(map(count_leaves, tree))
```

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.

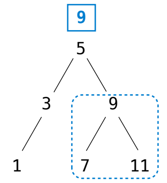


Proposal 1: A set is represented by a recursive list that contains no duplicate items.

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	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n)$

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

Exceptions are raised with a raise statement.

```
raise <expression>
```

<expression> must evaluate to an 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>
...
```

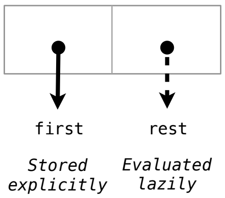
The <try suite> is executed first;

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(object):
    """A lazily computed recursive list."""
    class empty(object):
        def __repr__(self):
            return 'Stream.empty'
    empty = empty()

    def __init__(self, first, compute_rest=lambda: Stream.empty):
        assert callable(compute_rest), 'compute_rest must be callable.'
        self.first = first
        self._compute_rest = compute_rest
        self._rest = None
```

"Please don't reference directly"

```
    @property
    def rest(self):
        """Return the rest of the stream, computing it if necessary."""
        if self._compute_rest is not None:
            self._rest = self._compute_rest()
            self._compute_rest = None
        return self._rest

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()
```

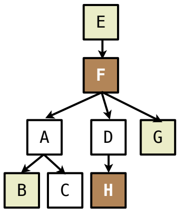
```
def map_stream(fn, s):
    if s is Stream.empty:
        return s
    def compute_rest():
        return map_stream(fn, s.rest)
    return Stream(fn(s.first),
                  compute_rest)
```

```
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)              > (define (abs x)
> (* pi 2)                          (if (< x 0)
6.28                                    (- x)
                                        x))
                                > (abs -3)
                                3
```

Lambda expressions evaluate to anonymous functions.

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

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.

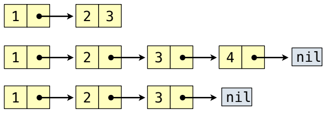
```
> (car '(a b c))
a
> (cdr '(a b c))
(b c)
```

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
```

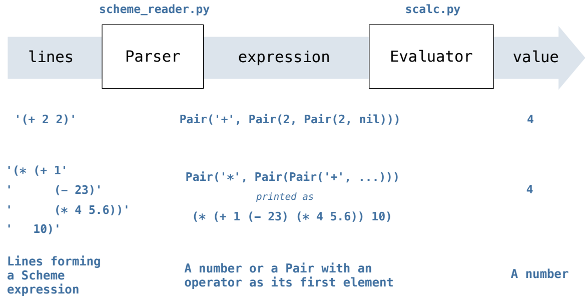
```
for <name> in <expression>:
    <suite>
```

1. Evaluate the header <expression>, which yields an iterable object.
2. For each element in that sequence, in order:
   A. Bind <name> to that element in the first frame of the current environment.
   B. Execute the <suite>.

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

```
>>> counts = [1, 2, 3]              >>> items = counts.__iter__()
>>> for item in counts:             >>> try:
        print(item)                         while True:
1                                               item = items.__next__()
2                                               print(item)
3                                       except StopIteration:
                                            pass
```

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



```
'(+ 2 2)'                 Pair('+', Pair(2, Pair(2, nil)))        4

'(* (+ 1'                 Pair('*', Pair(Pair('+', ...)))         4
'      (- 23)'                      printed as
'      (* 4 5.6))'        (* (+ 1 (- 23) (* 4 5.6)) 10)
'   10)'
```

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

A Scheme list is written as elements in parentheses:

(<element$_0$> <element$_1$> ... <element$_n$>)

A recursive Scheme list

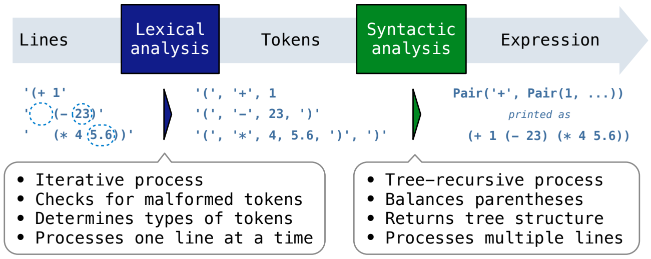
Each <element> can be a combination or atom (primitive).

(+ (* 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.



```
'(+ 1'                '(', '+', 1                         Pair('+', Pair(1, ...))
'   (- 23)'           '(', '-', 23, ')'                          printed as
'   (* 4 5.6))'       '(', '*', 4, 5.6, ')', ')'          (+ 1 (- 23) (* 4 5.6))
```

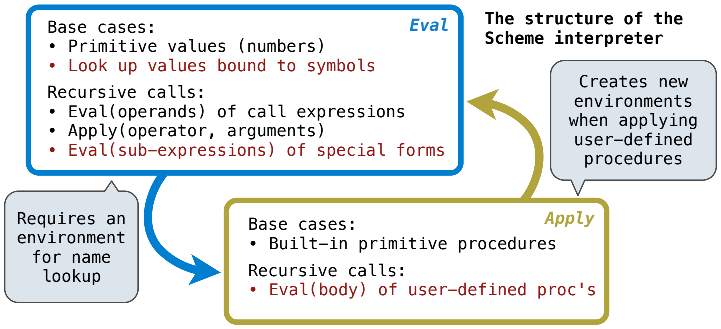
- Iterative process
- Checks for malformed tokens
- Determines types of tokens
- Processes one line at a time

- 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

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

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

Creates new environments when applying user-defined procedures

Requires an environment for name lookup

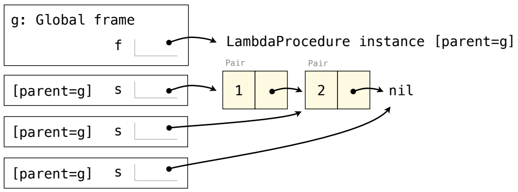
Base cases:
- Built-in primitive procedures

Recursive calls:
- Eval(body) of user-defined proc's

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)                  (define (length s)
  (if (= n 0) k                            (if (null? s) 0
    (factorial (- n 1)                       (+ 1 (length (cdr s)))))
               (* k n))))
```

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