CS 61A Structure and Interpretation of Computer Programs
Fall 2014

Midterm 2

INSTRUCTIONS

- You have 2 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 2 official 61A midterm study guides attached to the back of this exam.
- Mark your answers ON THE EXAM ITSELF. If you are not sure of your answer you may wish to provide a *brief* explanation.

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	Total
/12	/14	/8	/8	/8	/50

Blank Page

1. (12 points) Class Hierarchy

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

Whenever the interpreter would report an error, write ERROR. You *should* include any lines displayed before an error. *Reminder*: The interactive interpreter displays the `repr` string of the value of a successfully evaluated expression, unless it is `None`. Assume that you have started Python 3 and executed the following:

```
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting
class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'My job is to gather wealth'
class Proletariat(Worker):
    greeting = 'Comrade'
    def work(self, other):
        other.greeting = self.greeting + ' ' + other.greeting
        other.work() # for revolution
        return other
jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```

Expression	Interactive Output
5*5	25
1/0	ERROR
Worker().work()	
jack	
jack.work()	

Expression	Interactive Output
john.work()[10:]	
Proletariat().work(john)	
john.elf.work(john)	

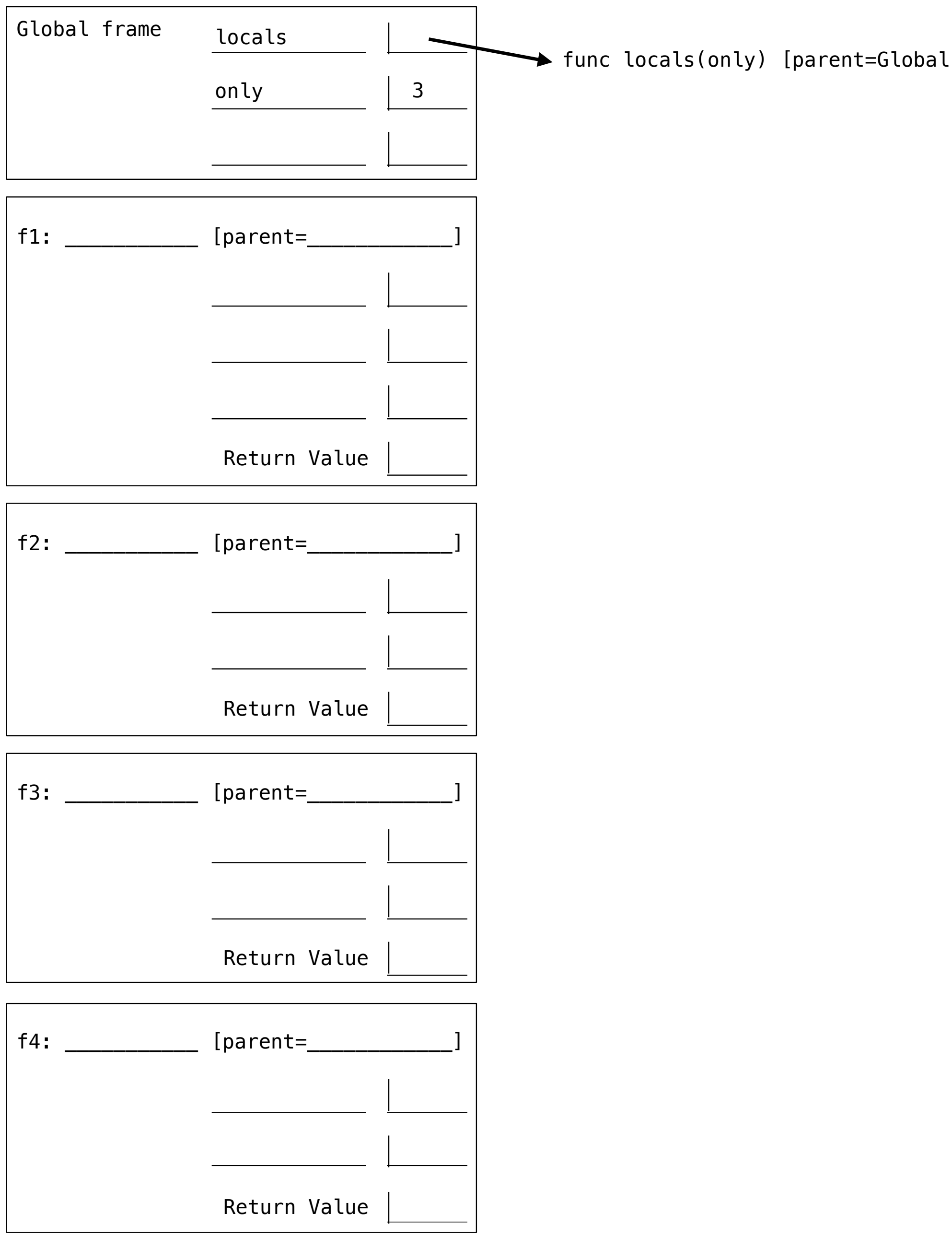
2. (14 points) Space

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

A complete answer will:

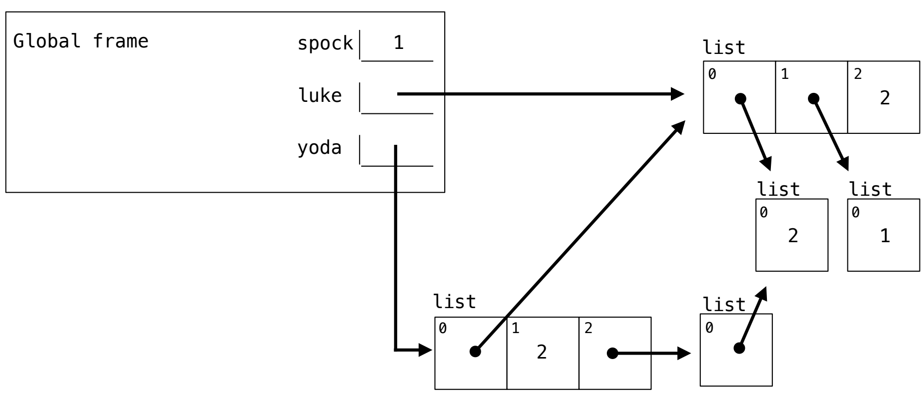
- Add all missing names and parent annotations to all local frames.
- Add all missing values created during execution.
- Show the return value for each local frame.

```
def locals(only):
    def get(out):
        nonlocal only
        def only(one):
            return lambda get: out
        out = out + 1
        return [out + 2]
    out = get(-only)
    return only

only = 3
earth = locals(only)
earth(4)(5)
```



(b) **(6 pt)** Fill in the blanks with the shortest possible expressions that complete the code in a way that results in the environment diagram shown. You can use only brackets, commas, colons, and the names `luke`, `spock`, and `yoda`. **You *cannot* use integer literals, such as `0`, in your answer!** You also cannot call any built-in functions or invoke any methods by name.

```
spock, yoda = 1, 2

luke = [_______________________________________________________]

yoda = 0

yoda = [_______________________________________________________]

yoda.append(___________________________________________________)
```



3. (8 points) This One Goes to Eleven

(a) (4 pt) Fill in the blanks of the implementation of `sixty_ones` below, a function that takes a `Link` instance representing a sequence of integers and returns the number of times that 6 and 1 appear consecutively.

```
def sixty_ones(s):
    """Return the number of times that 1 directly follows 6 in linked list s.

    >>> once = Link(4, Link(6, Link(1, Link(6, Link(0, Link(1))))))
    >>> twice = Link(1, Link(6, Link(1, once)))
    >>> thrice = Link(6, twice)
    >>> apply_to_all(sixty_ones, [Link.empty, once, twice, thrice])
    [0, 1, 2, 3]
    """
    if _______________________________________________________________:

        return 0

    elif _____________________________________________________________:


        return 1 + ___________________________________________________:

    else:

        return _______________________________________________________
```

(b) (4 pt) Fill in the blanks of the implementation of `no_eleven` below, a function that returns a list of all distinct length-`n` lists of ones and sixes in which 1 and 1 do not appear consecutively.

```
def no_eleven(n):
    """Return a list of lists of 1's and 6's that do not contain 1 after 1.

    >>> no_eleven(2)
    [[6, 6], [6, 1], [1, 6]]
    >>> no_eleven(3)
    [[6, 6, 6], [6, 6, 1], [6, 1, 6], [1, 6, 6], [1, 6, 1]]
    >>> no_eleven(4)[:4] # first half
    [[6, 6, 6, 6], [6, 6, 6, 1], [6, 6, 1, 6], [6, 1, 6, 6]]
    >>> no_eleven(4)[4:] # second half
    [[6, 1, 6, 1], [1, 6, 6, 6], [1, 6, 6, 1], [1, 6, 1, 6]]
    """
    if n == 0:

        return __________________________________________________________

    elif n == 1:

        return __________________________________________________________

    else:

        a, b = no_eleven(__________________), no_eleven(__________________)


        return [________________ for s in a] + [________________ for s in b]
```

4. (8 points) Tree Time

(a) **(4 pt)** A `GrootTree` g is a binary tree that has an attribute `parent`. Its parent is the `GrootTree` in which g is a branch. If a `GrootTree` instance is not a branch of any other `GrootTree` instance, then its `parent` is `BinaryTree.empty`.

`BinaryTree.empty` should not have a `parent` attribute. Assume that every `GrootTree` instance is a branch of at most one other `GrootTree` instance and not a branch of any other kind of tree.

Fill in the blanks below so that the `parent` attribute is set correctly. You may not need to use all of the lines. Indentation is allowed. You *should not* include any `assert` statements. Using your solution, the doctests for `fib_groot` should pass. The `BinaryTree` class appears on your study guide.

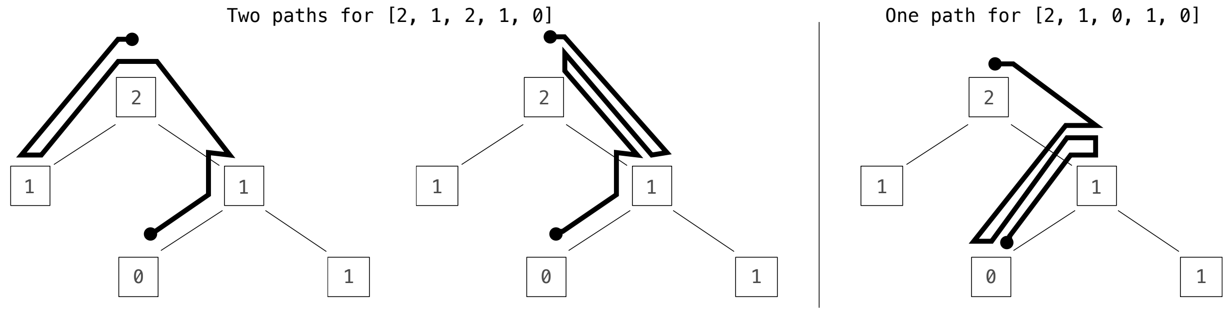
Hint: A picture of `fib_groot(3)` appears on the next page.

```
class GrootTree(BinaryTree):
    """A binary tree with a parent."""

    def __init__(self, entry, left=BinaryTree.empty, right=BinaryTree.empty):
        BinaryTree.__init__(self, entry, left, right)

        ____________________________________________________________________

        ____________________________________________________________________

        ____________________________________________________________________

        ____________________________________________________________________

        ____________________________________________________________________

        ____________________________________________________________________

def fib_groot(n):
    """Return a Fibonacci GrootTree.

    >>> t = fib_groot(3)
    >>> t.entry
    2
    >>> t.parent.is_empty
    True
    >>> t.left.parent.entry
    2
    >>> t.right.left.parent.right.parent.entry
    1
    """
    if n == 0 or n == 1:
        return GrootTree(n)
    else:
        left, right = fib_groot(n-2), fib_groot(n-1)
        return GrootTree(left.entry + right.entry, left, right)
```

(b) (4 pt) Fill in the blanks of the implementation of `paths`, a function that takes two arguments: a `GrootTree` instance `g` and a list `s`. It returns the number of paths through `g` whose entries are the elements of `s`. A path through a `GrootTree` can extend either to a branch or its `parent`.

You may assume that the `GrootTree` class is implemented correctly and that the list `s` is non-empty.

The two paths that have entries `[2, 1, 2, 1, 0]` in `fib_groot(3)` are shown below (left). The one path that has entries `[2, 1, 0, 1, 0]` is shown below (right).




```
def paths(g, s):
    """The number of paths through g with entries s.

    >>> t = fib_groot(3)
    >>> paths(t, [1])
    0
    >>> paths(t, [2])
    1
    >>> paths(t, [2, 1, 2, 1, 0])
    2
    >>> paths(t, [2, 1, 0, 1, 0])
    1
    >>> paths(t, [2, 1, 2, 1, 2, 1])
    8
    """

    if g is BinaryTree.empty ____________________________________________:

        return 0

    elif ________________________________________________________________:

        return 1

    else:

        xs = [______________________________________________________________]

        return sum([ ____________________________________________ for x in xs])
```

5. (8 points) Abstraction and Growth

(a) (6 pt) Your project partner has invented an abstract representation of a sequence called a `slinky`, which uses a `transition` function to compute each element from the previous element. A `slinky` explicitly stores only those elements that cannot be computed by calling `transition`, using a `starts` dictionary. Each entry in `starts` is a pair of an index key and an element value. See the doctests for examples.

Help your partner fix this implementation by crossing out as many lines as possible, but leaving a program that passes the doctests. Do not change the doctests. The program continues onto the following page.

```
def length(slinky):
    return slinky[0]
def starts(slinky):
    return slinky[1]
def transition(slinky):
    return slinky[2]

def slinky(elements, transition):
    """Return a slinky containing elements.

    >>> t = slinky([2, 4, 10, 20, 40], lambda x: 2*x)
    >>> starts(t)
    {0: 2, 2: 10}
    >>> get(t, 3)
    20
    >>> r = slinky(range(3, 10), lambda x: x+1)
    >>> length(r)
    7
    >>> starts(r)
    {0: 3}
    >>> get(r, 2)
    5
    >>> slinky([], abs)
    [0, {}, <built-in function abs>]
    >>> slinky([5, 4, 3], abs)
    [3, {0: 5, 1: 4, 2: 3}, <built-in function abs>]
    """
    starts = {}
    last = None
    for e in elements[1:]:
    for index in range(len(elements)):
        if not e:
        if index == 0:
            return [0, {}, transition]
        if last is None or e != transition(last):
        if e == 0 or e != transition(last):
        if index == 0 or elements[index] != transition(elements[index-1]):
            starts[index] = elements[index]
            starts[index] = elements.pop(index)
            starts[e] = transition(last)
            starts[e] = last
        last = e
    return [len(starts), starts, transition]
    return [len(elements), starts, transition]
    return [len(starts), elements, transition]
    return [len(elements), elements, transition]
```

```
def get(slinky, index):
    """Return the element at index of slinky."""
    if index in starts(slinky):
        return starts(slinky)[index]
    start = index
    start = 0
    f = transition(slinky)
    while start not in starts(slinky):
    while not f(get(start)) == index:
        start = start + 1
        start = start - 1
    value = starts(slinky)[start]
    value = starts(slinky)[0]
    value = starts(slinky)[index]
    while start < index:
    while value < index:
        value = f(value)
        value = value + 1
        start = start + 1
        start = start + index
    return value
    return f(value)
```

(b) **(2 pt)** Circle the Θ expression below that describes the number of operations required to compute `slinky(elements, transition)`, assuming that

- n is the initial length of `elements`,
- d is the final length of the `starts` dictionary created,
- the `transition` function requires constant time,
- the `pop` method of a list requires constant time,
- the `len` function applied to a `list` requires linear time,
- the `len` function applied to a `range` requires constant time,
- adding or updating an entry in a dictionary requires constant time,
- getting an element from a list by its index requires constant time,
- creating a list requires time that is proportional to the length of the list.

$\Theta(1)$ $\Theta(n)$ $\Theta(d)$ $\Theta(n^2)$ $\Theta(d^2)$ $\Theta(n \cdot d)$

Scratch Paper

Scratch Paper

Import statement

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

Assignment statement

Global frame: Name — pi — 3.1416 — Value; Binding

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)
```

Global frame
- mul → func mul(...) [parent=Global] — Built-in function
- square → func square(x) [parent=Global] — User-defined function

Intrinsic name of function called

f1: square [parent=Global] — Local frame
- x: -2 — Formal parameter bound to argument
- Return value: 4 — Return value is not a binding!

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

Global frame: mul, square

f1: square [parent=Global] — x 3, Return value 9

f2: square [parent=Global] — x 9, Return value 81

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

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.

```
208
mul(add(2, mul(4, 6)), add(3, 5))
  mul    26 add(2, mul(4, 6))        8 add(3, 5)
         add  2  24 mul(4, 6)        add  3  5
                 mul  4  6
```

Pure Functions

-2 → *abs*(number): → 2

2, 10 → *pow*(x, y): → 1024

Non-Pure Functions

-2 → *print*(...): → None

display "-2"

Defining:

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

Def statement; Formal parameter; Return expression; Body (*return statement*)

Call expression: square(2+2)

operand: 2+2
argument: 4

operator: square
function: func square(x)

Calling/Applying: 4 → *square*(x): return mul(x, x) → 16

Argument; Intrinsic name; Return value

Compound statement; Clause

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

Suite

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

1 statement,
3 clauses,
3 headers,
3 suites,
2 boolean contexts

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

def g(a):
    return a + y

result = f(1, 2)
```

"y" is not found

Error

"y" is not found

Global frame: f → func f(x, y) [parent=Global]; g → func g(a) [parent=Global]

f1: f [parent=Global] — x 1, y 2

f2: g [parent=Global] — a 1

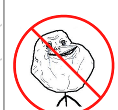
- An environment is a sequence of frames
- An environment for a non-nested function (no def within def) consists of one local frame, followed by the global frame

```
from operator import mul
def square(square):
    return mul(square, square)
square(4)
```

Global frame: mul, square

f1: square [parent=Global] — square 4, Return value 16

A call expression and the body of the function being called are evaluated in different environments



```
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1   # Zeroth and first Fibonacci numbers
    k = 1               # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```

```
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

The function bound to term gets called here

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

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

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

Evaluates to a function. No "return" keyword!

A function
with formal parameters x and y
that returns the value of "x * y"

Must be a single expression

```
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

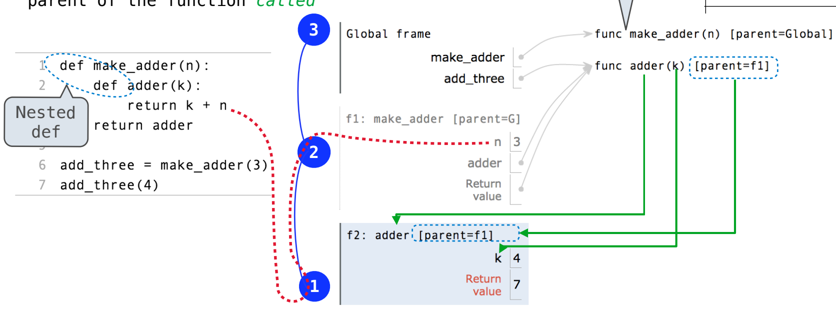
The name add_three is bound to a function

A local def statement

Can refer to names in the enclosing function

- Every user-defined **function** has a ***parent frame*** (often global)
- The parent of a **function** is the frame in which it was *defined*
- Every local **frame** has a *parent* ***frame*** (often global)
- The parent of a **frame** is the parent of the function *called*

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



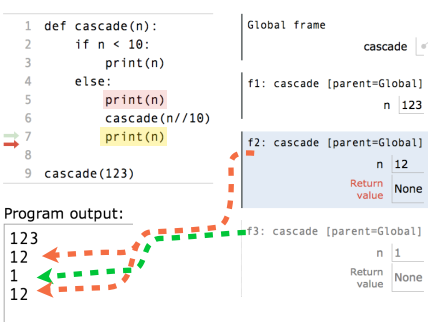

```
def curry2(f):
    """Returns a function g such that g(x)(y) returns f(x, y)."""
    def g(x):
        def h(y):
            return f(x, y)
        return h
    return g
```

Currying: Transforming a multi-argument function into a single-argument, higher-order function.

Anatomy of a recursive function:

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

```
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = n // 10, n % 10
        return sum_digits(all_but_last) + last
```



- Each **cascade** frame is from a different call to **cascade.**
- Until the Return value appears, that call has not completed.
- Any statement can appear before or after the recursive call.

1
12
123
1234
123
12
1

```
def inverse_cascade(n):
    grow(n)
    print(n)
    shrink(n)

def f_then_g(f, g, n):
    if n:
        f(n)
        g(n)

grow =   lambda n: f_then_g(grow,  print,  n//10)
shrink = lambda n: f_then_g(print, shrink, n//10)
```

n:	0,	1,	2,	3,	4,	5,	6,	7,	8,
fib(n):	0,	1,	1,	2,	3,	5,	8,	13,	21,

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

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

VS

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

- Both create a function with the same domain, range, and behavior.
- Both functions have as their parent the environment in which they were defined.
- Both bind that function to the name square.
- Only the def statement gives the function an intrinsic name.

When a function is defined:

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

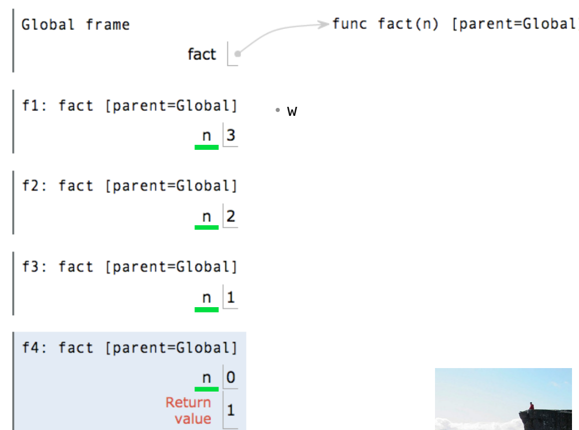
   f1: make_adder   func adder(k) [parent=f1]

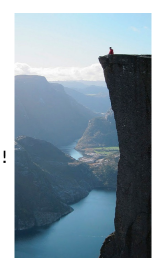
3. Bind <name> to the **function value** in the current frame (which is the first frame of the current environment).

When a function is called:

1. Add a **local frame**, titled with the <name> of the function being called.
2. Copy the parent of the function to the **local frame**: [parent=<label>]
3. Bind the <formal parameters> to the arguments in the **local frame**.
4. Execute the body of the function in the environment that starts with the **local frame**.

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

fact(3)
```





Is fact implemented correctly?
1. Verify the base case.
2. Treat fact as a functional abstraction!
3. Assume that fact(n-1) is correct.
4. Verify that fact(n) is correct, assuming that fact(n-1) correct.

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

```
def count_partitions(n, m):
    if n == 0:
        return 1
    elif n < 0:
        return 0
    elif m == 0:
        return 0
    else:
        with_m = count_partitions(n-m, m)
        without_m = count_partitions(n, m-1)
        return with_m + without_m
```

```
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

Numeric types in Python:

```
>>> type(2)
<class 'int'>
```
Represents integers exactly

```
>>> type(1.5)
<class 'float'>

>>> type(1+1j)
<class 'complex'>
```
Represents real numbers approximately

Functional pair implementation:

```
def pair(x, y):
    """Return a functional pair."""
    def get(index):
        if index == 0:
            return x
        elif index == 1:
            return y
    return get
```
This function represents a pair

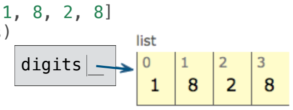
Constructor is a higher-order function

```
def select(p, i):
    """Return element i of pair p."""
    return p(i)
```
Selector defers to the object itself

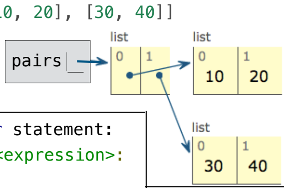
```
>>> p = pair(1, 2)
>>> select(p, 0)
1
>>> select(p, 1)
2
```

Lists:

```
>>> digits = [1, 8, 2, 8]
>>> len(digits)
4
>>> digits[3]
8
```




```
>>> [2, 7] + digits * 2
[2, 7, 1, 8, 2, 8, 1, 8, 2, 8]

>>> pairs = [[10, 20], [30, 40]]
>>> pairs[1]
[30, 40]
>>> pairs[1][0]
30
```



Executing a for statement:

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

1. Evaluate the header <expression>, which must yield an iterable value (a sequence)
2. For each element in that sequence, in order:
   A. Bind <name> to that element in the current frame
   B. Execute the <suite>

Unpacking in a for statement:

A sequence of fixed-length sequences

```
>>> pairs=[[1, 2], [2, 2], [3, 2], [4, 4]]
>>> same_count = 0
```

A name for each element in a fixed-length sequence

```
>>> for x, y in pairs:
...     if x == y:
...         same_count = same_count + 1

>>> same_count
2
```

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

range(-2, 2)

Length: ending value - starting value
Element selection: starting value + index

```
>>> list(range(-2, 2))
[-2, -1, 0, 1]
```
List constructor

```
>>> list(range(4))
[0, 1, 2, 3]
```
Range with a 0 starting value

Membership:

```
>>> digits = [1, 8, 2, 8]
>>> 2 in digits
True
>>> 1828 not in digits
True
```

Slicing:

```
>>> digits[0:2]
[1, 8]
>>> digits[1:]
[8, 2, 8]
```
Slicing creates a new object

List comprehensions:

[<map exp> for <name> in <iter exp> if <filter exp>]

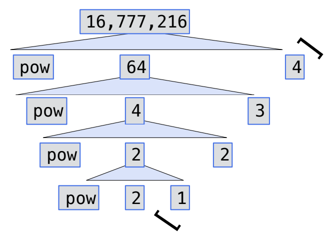
Short version: [<map exp> for <name> in <iter exp>]

A combined expression that evaluates to a list using this evaluation procedure:
1. Add a new frame with the current frame as its parent
2. Create an empty *result list* that is the value of the expression
3. For each element in the iterable value of <iter exp>:
   A. Bind <name> to that element in the new frame from step 1
   B. If <filter exp> evaluates to a true value, then add the value of <map exp> to the result list

```
def apply_to_all(map_fn, s):
    """Apply map_fn to each element of s.

    >>> apply_to_all(lambda x: x*3, range(5))
    [0, 3, 6, 9, 12]
    """
    return [map_fn(x) for x in s]
```
0, 1, 2, 3, 4 → λx: x*3 → 0, 3, 6, 9, 12

```
def keep_if(filter_fn, s):
    """List elements x of s for which
    filter_fn(x) is true.

    >>> keep_if(lambda x: x>5, range(10))
    [6, 7, 8, 9]
    """
    return [x for x in s if filter_fn(x)]
```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9 → λx: x>5 → 6, 7, 8, 9

```
def reduce(reduce_fn, s, initial):
    """Combine elements of s pairwise using reduce_fn,
    starting with initial.
    """
    r = initial
    for x in s:
        r = reduce_fn(r, x)
    return r

reduce(pow, [1, 2, 3, 4], 2)
```



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

$R(n) = \Theta(f(n))$ means that there are positive constants k_1 and k_2 such that $k_1 \cdot f(n) \le R(n) \le k_2 \cdot f(n)$ for all n larger than some m

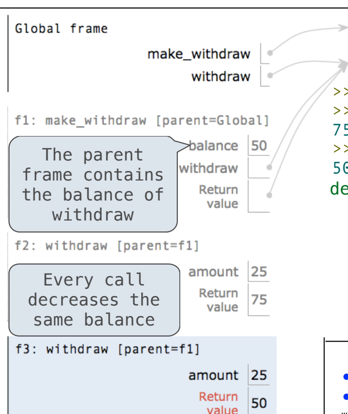
$\Theta(b^n)$ Exponential growth. Recursive fib takes $\Theta(\phi^n)$ steps, where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61828$
Incrementing the problem scales R(n) by a factor

$\Theta(n^2)$ Quadratic growth. E.g., overlap
Incrementing n increases R(n) by the problem size n

$\Theta(n)$ Linear growth. E.g., factors or exp

$\Theta(\log n)$ Logarithmic growth. E.g., exp_fast
Doubling the problem only increments R(n)

$\Theta(1)$ Constant. The problem size doesn't matter




```
>>> withdraw = make_withdraw(100)
>>> withdraw(25)
75
>>> withdraw(25)
50
def make_withdraw(balance):
    def withdraw(amount):
        nonlocal balance
        if amount > balance:
            return 'No funds'
        balance = balance - amount
        return balance
    return withdraw
```

Status	x = 2	Effect
•No nonlocal statement •"x" **is not** bound locally		Create a new binding from name "x" to number 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 environment
•nonlocal x •"x" **is** bound in a non-local frame		Re-bind "x" to 2 in the first non-local frame of the current environment in which "x" 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

Strings as sequences:

```
>>> city = 'Berkeley'
>>> len(city)
8
>>> city[3]
'k'
>>> 'here' in "Where's Waldo?"
True
>>> 234 in [1, 2, 3, 4, 5]
False
>>> [2, 3, 4] in [1, 2, 3, 4]
False
```

List & dictionary mutation:

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

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

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

Identity:
<exp0> is <exp1>
evaluates to True if both <exp0> and <exp1> evaluate to the same object
Equality:
<exp0> == <exp1>
evaluates to True if both <exp0> and <exp1> evaluate to equal values
Identical objects are always equal values

You can copy a list by calling the list constructor or slicing the list from the beginning to the end.

Constants: Constant terms do not affect the order of growth of a process
$\Theta(n)$ $\Theta(500 \cdot n)$ $\Theta(\frac{1}{500} \cdot n)$

Logarithms: The base of a logarithm does not affect the order of growth of a process
$\Theta(\log_2 n)$ $\Theta(\log_{10} n)$ $\Theta(\ln n)$

Nesting: When an inner process is repeated for each step in an outer process, multiply the steps in the outer and inner processes to find the total number of steps

```
def overlap(a, b):
    count = 0
    for item in a:
        if item in b:
            count += 1
    return count
```
Outer: length of a
Inner: length of b

If a and b are both length **n**, then overlap takes $\Theta(n^2)$ steps

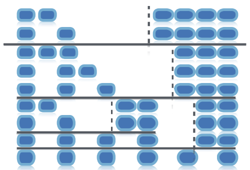
Lower-order terms: The fastest-growing part of the computation dominates the total
$\Theta(n^2)$ $\Theta(n^2 + n)$ $\Theta(n^2 + 500 \cdot n + \log_2 n + 1000)$

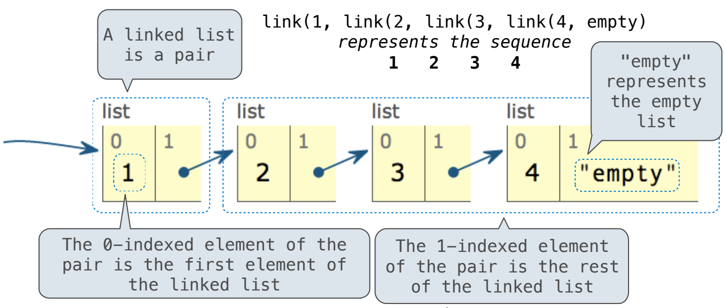
CS 61A Midterm 2 Study Guide — Page 2

Linked list data abstraction:

```
empty = 'empty'

def link(first, rest):
    return [first, rest]

def first(s):
    return s[0]

def rest(s):
    return s[1]

def len_link(s):
    x = 0
    while s != empty:
        s, x = rest(s), x+1
    return x

def getitem_link(s, i):
    while i > 0:
        s, i = rest(s), i - 1
    return first(s)

def extend(s, t):
    assert is_link(s) and is_link(t)
    if s == empty:
        return t
    else:
        return link(first(s), extend(rest(s), t))

def apply_to_all_link(f, s):
    if s == empty:
        return s
    else:
        return link(f(first(s)), apply_to_all_link(f, rest(s)))
```

```
def partitions(n, m):
    """Return a linked list of partitions
    of n using parts of up to m.
    Each partition is a linked list.
    """
    if n == 0:
        return link(empty, empty)
    elif n < 0:
        return empty
    elif m == 0:
        return empty
    else:
        # Do I use at least one m?
        yes = partitions(n-m, m)
        no = partitions(n, m-1)
        add_m = lambda s: link(m, s)
        yes = apply_to_all_link(add_m, yes)
        return extend(yes, no)
```





The result of calling **repr** on a value is what Python prints in an interactive session

The result of calling **str** on a value is what Python prints using the **print** function

```
>>> 12e12
12000000000000.0
>>> print(repr(12e12))
12000000000000.0
```

```
>>> print(today)
2014-10-13
```

Memoization:

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

str and **repr** are both polymorphic; they apply to any object
repr invokes a zero-argument method __repr__ on its argument

```
>>> today.__repr__()
'datetime.date(2014, 10, 13)'
```

```
>>> today.__str__()
'2014-10-13'
```

```
class Link:
    empty = ()
    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest
    def __getitem__(self, i):
        if i == 0:
            return self.first
        else:
            return self.rest[i-1]
    def __len__(self):
        return 1 + len(self.rest)
```

Some zero length sequence

Yes, this call is recursive

Sequence abstraction special names:

__getitem__	Element selection []
__len__	Built-in len function

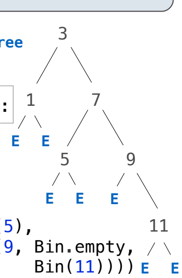
```
class Tree:
    def __init__(self, entry, branches=()):
        self.entry = entry
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)
```

Built-in isinstance function: returns True if branch has a class that *is* **or** *inherits from* Tree

```
class BinaryTree(Tree):
    empty = Tree(None)
    empty.is_empty = True
    def __init__(self, entry, left=empty, right=empty):
        Tree.__init__(self, entry, (left, right))
        self.is_empty = False
    @property
    def left(self):
        return self.branches[0]
    @property
    def right(self):
        return self.branches[1]
```

E: An empty tree

```
Bin = BinaryTree
t = Bin(3, Bin(1),
           Bin(7, Bin(5),
                  Bin(9, Bin.empty,
                         Bin(11))))
```



Python object system:

Idea: All bank accounts have a **balance** and an account **holder**; the **Account** class should add those attributes to each of its instances

A new instance is created by calling a class

```
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
```

An account instance

balance: 0 holder: 'Jim'

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

__init__ is called a constructor

self should always be bound to an instance of the Account class or a subclass of Account

```
class Account:
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
    def withdraw(self, amount):
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance
```

Function call: all arguments within parentheses

```
>>> type(Account.deposit)
<class 'function'>
>>> type(a.deposit)
<class 'method'>
```

Method invocation: One object before the dot and other arguments within parentheses

```
>>> Account.deposit(a, 5)
10
>>> a.deposit(2)
12
```

Call expression

Dot expression

<expression> . <name>

The <expression> can be any valid Python expression.
The <name> must be a simple name.
Evaluates to the value of the attribute looked up by <name> in the object that is the value of the <expression>.

To evaluate a dot expression:
1. Evaluate the <expression> to the left of the dot, which yields the object of the dot expression
2. <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

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

Account class attributes

interest: ~~0.02~~ ~~0.04~~ 0.05
(withdraw, deposit, __init__)

Instance attributes of jim_account

balance: 0
holder: 'Jim'
interest: 0.08

Instance attributes of tom_account

balance: 0
holder: 'Tom'

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

```
>>> jim_account.interest = 0.08
>>> jim_account.interest
0.08
>>> tom_account.interest
0.04
>>> Account.interest = 0.05
>>> tom_account.interest
0.05
>>> jim_account.interest
0.08
```

```
class CheckingAccount(Account):
    """A bank account that charges for withdrawals."""
    withdraw_fee = 1
    interest = 0.01
    def withdraw(self, amount):
        return Account.withdraw(self, amount + self.withdraw_fee)
                                       or
        return super().withdraw(      amount + self.withdraw_fee)
```

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('Tom')  # Calls Account.__init__
>>> ch.interest     # Found in CheckingAccount
0.01
>>> ch.deposit(20)  # Found in Account
20
>>> ch.withdraw(5)  # Found in CheckingAccount
14
```