CS 61A Structure and Interpretation of Computer Programs

Fall 2011 Final Exam

INSTRUCTIONS

- You have 3 hours to complete the exam.
- The exam is closed book, closed notes, closed computer, closed calculator, except a one-page crib sheet of your own creation and the three official 61A exam study guides, which are 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	
For which assignments do you have unresolved regrade requests?	
All the work on this exam is my own. **(please sign)**	

For staff use only

Q. 1	Q. 2	Q. 3	Q. 4	Q. 5	Q. 6	Total
/12	/16	/12	/18	/10	/12	/80

1. (12 points) What Would Python Print?

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

```
def oracle(a, b, c):
    if a == 42:
        return b(a)
    return c

class Big(object):
    def medium(self, d):
        def small(e):
            nonlocal d
            if e > 0:
                d = d + self.medium(e)(-e)
            return d
        return small

class Huge(Big):
    def medium(self, d):
        return Big.medium(self, d+1)
```

For each of the following expressions, write the `repr` string (i.e., the string printed by Python interactively) of the value *to which it evaluates* in the current environment. If evaluating the expression causes an error, write "Error." Any changes made to the environment by the expressions below *will affect* the subsequent expressions.

(a) (2 pt) `oracle(41, lambda x: 1/0, 'blue pill')`

(b) (2 pt) `oracle(42, lambda x: 'red pill', 1/0)`

(c) (2 pt) `Big.medium(self, -2)(-1)`

(d) (2 pt) `Big().medium(1)(Big().medium(2)(3))`

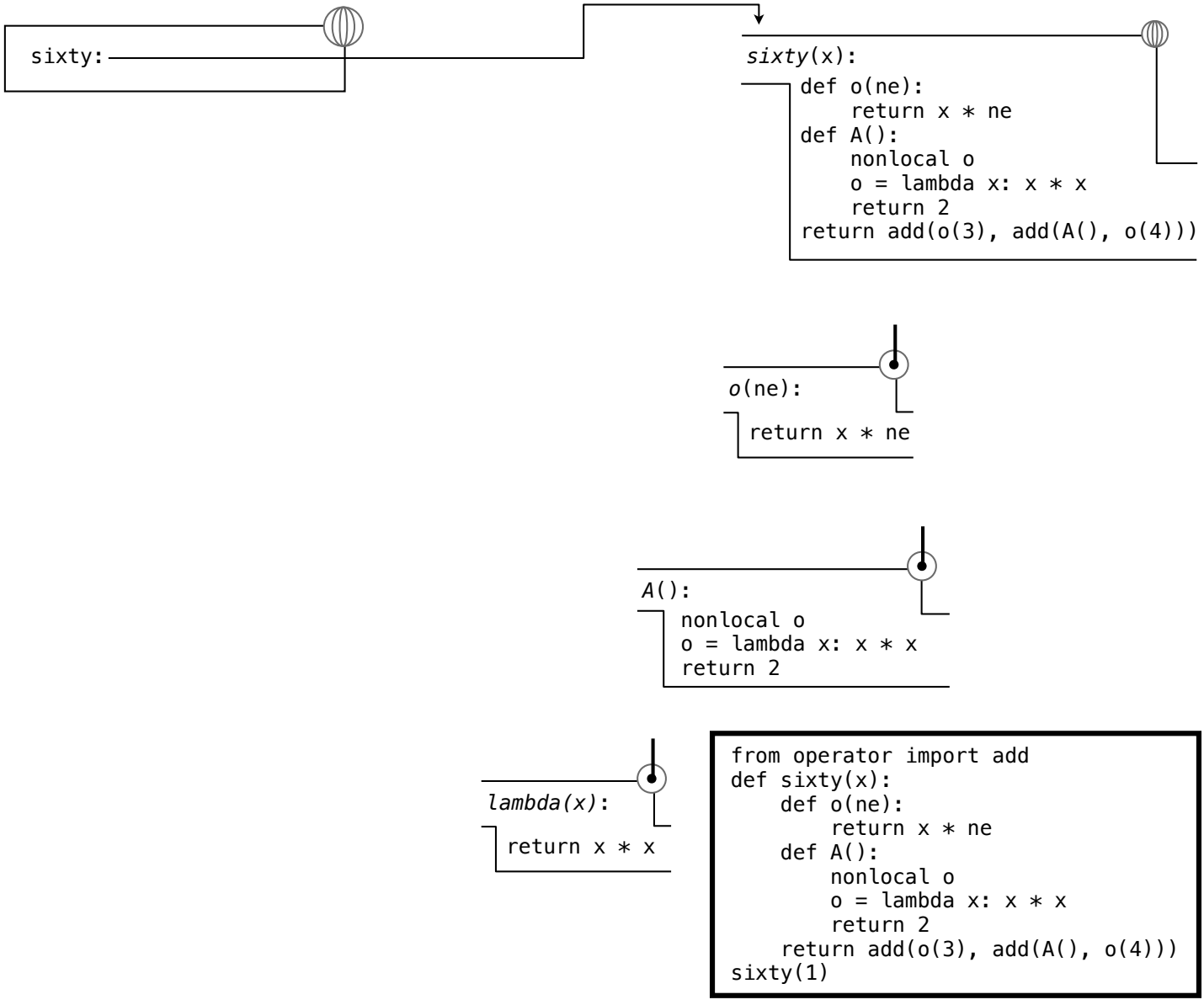
(e) (2 pt) `Huge().medium(4)(5)`

(f) (2 pt) `Big() == Huge()`

2. (16 points) Environment Diagrams.

(a) (6 pt) Complete the environment diagram for the program in the box below. You **do not** need to draw an expression tree. A complete answer will:

- Complete all missing arrows. Arrows to the global frame can be abbreviated by small globes.
- Add all local frames created by applying user-defined functions.
- Add all missing names in frames.
- Add all **final** values referenced by frames.

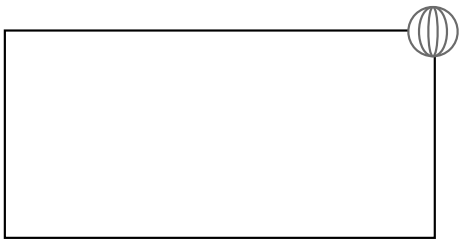


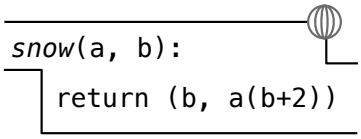

```
from operator import add
def sixty(x):
    def o(ne):
        return x * ne
    def A():
        nonlocal o
        o = lambda x: x * x
        return 2
    return add(o(3), add(A(), o(4)))
sixty(1)
```

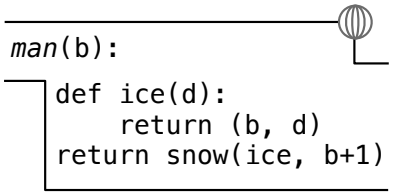
(b) (2 pt) What value is returned by evaluating `sixty(1)`? If evaluation causes an error, write "Error."

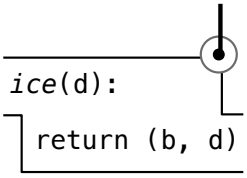
(c) (6 pt) Complete the environment diagram for the program in the box below. Assume Python's normal **lexical scoping** rules. You **do not** need to draw an expression tree. A complete answer will:

- Complete all missing arrows. Arrows to the global frame can be abbreviated by small globes.
- Add all local frames created by applying user-defined functions.
- Add all missing names in frames.
- Add all **final** values referenced by frames. Represent tuple values using box-and-pointer notation.










```
def snow(a, b):
    return (b, a(b+2))

def man(b):
    def ice(d):
        return (b, d)
    return snow(ice, b+1)

e = man(2)
```

(d) (2 pt) If Python were instead a **dynamically scoped** language, what would be the value bound to `e` in the global frame after this program was executed? Write the repr string of this value.

3. (12 points) Pyth-On Vacation.

Finish implementing this account object in Logo. Unlike a Python `Account` from lecture, the balance of this Logo account is stored as a global variable called `bal`.

The desired behavior of the account is to accept `deposit` and `withdraw` messages. `Deposit` increases the balance by an amount, while `withdraw` reduces the amount if funds are available.

```
? make "john_account make_account 100
? print invoke "withdraw :john_account 40
60
? print invoke "withdraw :john_account 70
insufficient_funds
? print invoke "deposit :john_account 20
80
? print invoke "withdraw :john_account 70
10
```

Invoking a method calls the corresponding named procedure. The `deposit` and `withdraw` procedures are implemented below.

```
to deposit :amt
  make "bal :bal + :amt output :bal
end

to withdraw :amt
  ifelse :amt > :bal [output "insufficient_funds] [make "bal :bal - :amt output :bal]
end
```

(a) (6 pt) Add punctuation and the word `run` to `make_account` and `invoke` so that the account object behaves as specified. You may add quotation marks, square brackets, colons, parentheses, and the word `run`, but **no other words**. *Do not remove any words. Multiple calls to* `run` *may be needed.*

```
to make_account :balance

                make              bal              balance

                output              sentence              message              amount

end

to invoke :message :account :amount

                output              account

end
```

(b) (6 pt) The D33P language includes three types of tokens: open parentheses, close parentheses, and integers. An expression is well-formed if it contains balanced parentheses, and each integer correctly indicates its depth: the number of nested sets of parentheses that surround that integer.

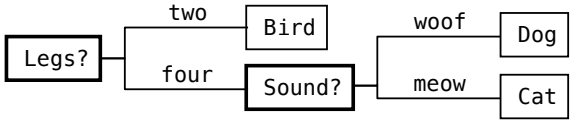
Implement `correct_depth`, which takes a list of tokens as input and returns True if and only if a prefix of the input is a well-formed D33P expression. Assume that the input contains a balanced set of nested parentheses with single-digit positive integers surrounded by parentheses. You only need to check that the integers indicate the correct depths.

Do not change any of the code that is provided. You may not use any `def` *statements or* `lambda` *expressions.*

```
def correct_depth(s, depth=0):
    """Return whether a prefix of list s is a well-formed D33P expression.

    >>> list('(1)')
    ['(', '1', ')']
    >>> correct_depth(list('(1)'))
    True
    >>> correct_depth(list('(2)'))
    False
    >>> correct_depth(list('((2)((3)))'))
    True
    >>> correct_depth(list('((2)(3))'))
    False
    >>> correct_depth(list('((3)(2))'))
    False
    >>> correct_depth(list('(((3)((4))(3))(2)((3)))'))
    True
    """
    first = s.pop(0)
    if first != '(':
        return depth==int(first)
```

4. (18 points) Interpreters. *It is possible to complete each part of this question without completing the others.*

The *Decider* language applies *decisions*, which are tree-structured rules that allow complex classification schemes to be decomposed into hierarchies of lookups. For instance, the problem of classifying household pets might use the following "decision" to distinguish among birds, cats, and dogs.



To decide what sort of animal something is, we look up its *features*. We first check whether it has two legs or four. If it has four legs, we check what sound it makes. Applying a decision always begins at the root (left).

All values in *Decider* are dictionaries with string-valued keys, such as the dog `fido` and the bird `tweety`.

```
fido =   {'name': 'fido',   'legs': 'four', 'sound': 'woof'}
tweety = {'name': 'tweety', 'legs': 'two'}
```

A *decision* is a dictionary that has `'?'` as a key bound to its decision feature. A decision also contains options. The values of options can be decisions, forming a tree. The `animals` decision below matches the diagram above.

```
four_legged = {'?': 'sound',   'woof': {'kind': 'dog'},   'meow': {'kind': 'cat'}}
animals =     {'?': 'legs',    'two': {'kind': 'bird'},   'four': four_legged}
```

(a) (4 pt) The `decide` function takes a `decision` and some `features` (both dictionaries). First, it finds the decision feature `f`. Next, it looks up the option for that feature stored in `features`. Finally, it returns the value for that option. Fill in the *four* missing names in the blanks in `decide`, which raises a `DecisionError` whenever the `decision` cannot be applied to the `features`.

```
def decide(decision, features):
    """Apply a decision to some features, both of which are dictionaries.

    >>> decide(four_legged, fido)
    {'kind': 'dog'}
    >>> decide(animals, fido)  # Returns the four_legged decision
    {'meow': {'kind': 'cat'}, 'woof': {'kind': 'dog'}, '?': 'sound'}
    """
    if '?' not in decision:
        raise DecisionError('Decision has no ? -- not a decision.')


    f = ______________________________['?']
    if f not in features:
        raise DecisionError('Features does not contain the decision feature.')


    option = ______________________________[f]
    if option not in decision:
        raise DecisionError('Decision does not contain selected option.')


    return ______________________________ [ ______________________________ ]

class DecisionError(Exception):
    """An error raised while applying a decision."""
```

(b) (6 pt) The `animals` decision is a depth-two tree that contains `four_legged` as the value for its `'four'` option. The `decider_apply` function repeatedly applies the result of a decision to some features until it finds a result that cannot be applied. In this way, it traverses a tree-structured decision.

Implement `decider_apply`, which must call `decide` within a `try` statement. This function repeatedly applies the result of `decide` to the `features`. It returns the result of the last successful call to `decide`. If `decision` cannot successfully be applied to `features` at all, return `decision`.

You may not use any `def` *statements or* `lambda` *expressions in your implementation.*

```
def decider_apply(decision, features):
    """Traverse the decision by applying sub-decisions to features.

    >>> decider_apply(four_legged, fido)
    {'kind': 'dog'}
    >>> decider_apply(animals, fido)
    {'kind': 'dog'}
    >>> decider_apply(animals, tweety)
    {'kind': 'bird'}
    >>> decider_apply(fido, tweety)  # fido is not a decision, so return it
    {'sound': 'woof', 'legs': 'four', 'name': 'fido'}
    """
```

(c) (6 pt) All expressions in *Decider* are dictionaries. *Compound expressions* contain the keys `'operator'` and `'operand'`. Two compound expressions, `big_pet_exp` and `kind_of_pet_exp`, appear below.

```
pets = {'?': 'size', 'small': tweety, 'large': fido}
big_pet_exp = {'operator': pets, 'operand': {'size': 'large'}}
kind_of_pet_exp = {'operator': animals, 'operand': big_pet_exp}
```

To evaluate a compound expression, evaluate its operator, evaluate its operand, then apply the `decision` dictionary that is the value of the operator to the `features` dictionary that is the value of the operand. All expressions that are not compound expressions are self-evaluating.

Complete the implementation of `decider_eval` by filling in the missing expressions.

```
def decider_eval(exp):
    """Evaluate a Decider expression.

    >>> decider_eval(fido)
    {'sound': 'woof', 'legs': 'four', 'name': 'fido'}
    >>> decider_eval(big_pet_exp)
    {'sound': 'woof', 'legs': 'four', 'name': 'fido'}
    >>> decider_eval(kind_of_pet_exp)
    {'kind': 'dog'}

    >>> decisions = {'?': '?', 'legs': animals, 'size': pets}
    >>> sub_exp = {'operator': decisions, 'operand': {'?': 'legs'}}
    >>> nested_exp = {'operator': sub_exp, 'operand': fido}
    >>> decider_eval(nested_exp)
    {'kind': 'dog'}
    """



    if ____________________________________________________________________:



        decision = _____________________________________________



        features = _____________________________________________

        return decider_apply(decision, features)

    else:

        return exp
```

(d) (2 pt) To what value does the following Python expression evaluate? Write its repr string.

```
decider_eval({'operator': pets, 'operand': {'size': 'small'}})
```

5. (10 points) Concurrency.

The following two statements are executed in parallel in a shared environment in which `x` is bound to `3`.

```
>>> x = x + 1
>>> x = x + 2 * x
```

(a) (2 pt) List all possible values of `x` at the end of execution.

(b) (2 pt) From the values you listed in (a), list all possible *correct* values of `x` at the end of execution.

The following program manages concurrent access to a shared dictionary called `grades` that maps student names to their 61A grades. The list `roster` contains the names of all students who have assigned grades.

```
grades = {}
roster = []
grades_lock, roster_lock = Lock(), Lock()

 def add_grade(name, grade):
     grades_lock.acquire()
     grades[name] = grade
     roster_lock.acquire()
     if name not in roster:
         roster.append(name)
     roster_lock.release()
     grades_lock.release()
```

```
def remove_grade(name):
    roster_lock.acquire()
    grades_lock.acquire()
    if name in grades:
        grades.pop(name)
    if name in roster:
        roster.remove(name)
    grades_lock.release()
    roster_lock.release()
```

For each set of statements (left and right) executed concurrently below, circle all of the problems that *may* occur, or circle *None of these are possible* if none of the listed problems may possibly occur.

(c) (2 pt)

`>>> add_grade('eric_k', 1)`	`>>> add_grade('steven', 2)`

(A) Deadlock
(B) Both processes simultaneously write to `grades` or `roster`
(C) None of these are possible

(d) (2 pt)

`>>> add_grade('stephanie', 3)`	`>>> roster.append('eric_t')` `>>> grades['eric_t'] = 4`

(A) Deadlock
(B) Both processes simultaneously write to `grades` or `roster`
(C) None of these are possible

(e) (2 pt)

`>>> add_grade('phill', 5)`	`>>> add_grade('aki', 6)` `>>> remove_grade('aki')`

(A) Deadlock
(B) Both processes simultaneously write to `grades` or `roster`
(C) None of these are possible

6. (12 points) Iterators and Streams.

(a) (4 pt) The generator function `unique` takes an iterable argument and returns an iterator over all the unique elements of its input in the order that they first appear. Implement `unique` **without** using a `for` statement. *You may not use any* `def`*,* `for`*, or* `class` *statements or* `lambda` *expressions.*

```
def unique(iterable):
    """Return an iterator over the unique elements of an iterable input.

    >>> list(unique([1, 3, 2, 2, 5, 3, 4, 1]))
    [1, 3, 2, 5, 4]
    """
```

The function `sum_grid` takes a stream of streams `s` and a length `n` and returns the sum of the first `n` elements of the first `n` streams in `s`. `Stream` *and* `make_integer_stream` *are defined in your study guide.*

```
def sum_grid(s, n):
    total = 0
    for _ in range(n):
        t = s.first
        for _ in range(n):
            total = total + t.first
            t = t.rest
        s = s.rest
    return total

def make_integer_grid(first=1):
    def compute_rest():
        return make_integer_grid(first+1)
    return Stream(make_integer_stream(first), compute_rest)
```

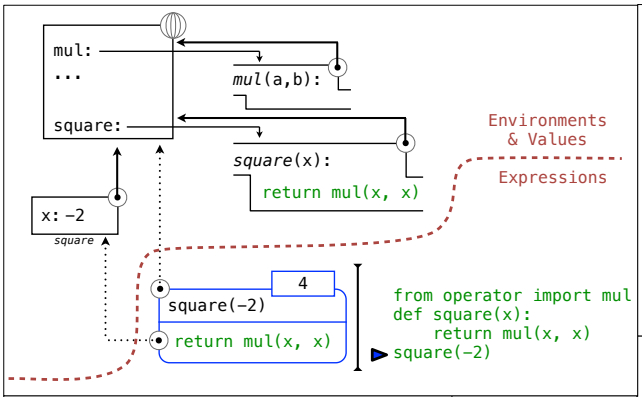
(b) (2 pt) What is the value of `sum_grid(make_integer_grid(1), 4)`?

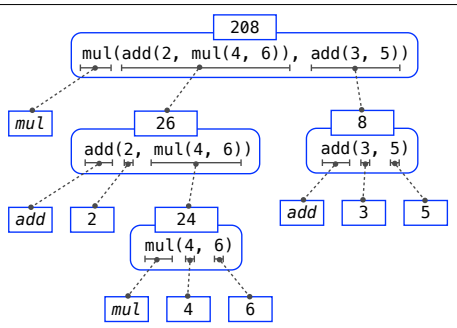
(c) (2 pt) Define a mathematical function $f(n)$ such that evaluating `sum_grid(make_integer_grid(1), n)` performs $\Theta(f(n))$ addition operations.

$f(n) =$

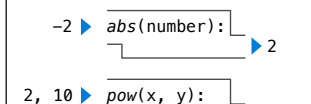
(d) (4 pt) The function `repeating` returns a `Stream` of integers that begins with `start` and increments each successive value until `stop` would be reached, at which point it returns to `start` and repeats.
Cross out lines from the body of `repeating` so that it correctly implements this behavior.

```
def repeating(start, stop):
        """Return a stream of integers that repeats the range(start, stop).

        >>> s = repeating(3, 6)
        >>> s.first, s.rest.first, s.rest.rest.first, s.rest.rest.rest.first,
        (3, 4, 5, 3)
        >>> s.rest.rest.rest.rest.first
        4
        """
        def make_stream(current):

                def compute_rest(next):

                def compute_rest():

                        nonlocal start

                        next = current+1

                        next = current % (stop-start)

                        if next > stop:

                        if next == stop:

                                next = start

                                start = next

                        return make_stream(start)

                        return make_stream(next)

                        return Stream(current, make_stream(start))

                        return Stream(current, make_stream(next))

                return Stream(current, compute_rest)

                return Stream(current, compute_rest())

                return compute_rest

                return compute_rest()

        return make_stream(start)
```

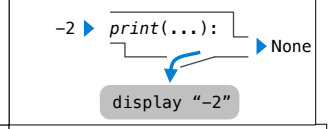


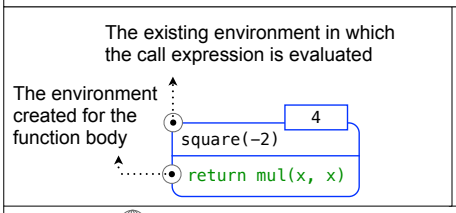


Pure Functions

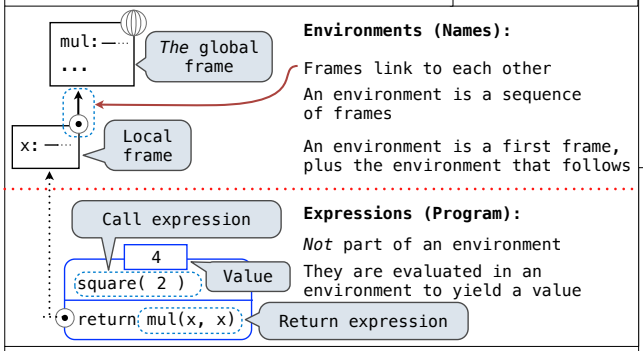


Non-Pure Functions





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



Environments (Names):

Frames link to each other

An environment is a sequence of frames

An environment is a first frame, plus the environment that follows

Expressions (Program):

Not part of an environment

They are evaluated in an environment to yield a value

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 that extends the environment with which the function is associated.
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. Associate that function with 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>.

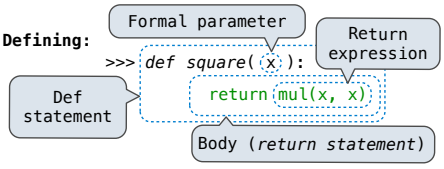
Evaluation rule for not expressions:

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

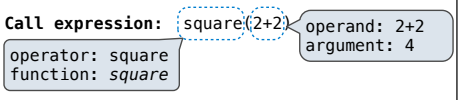
Execution rule for while statements:

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

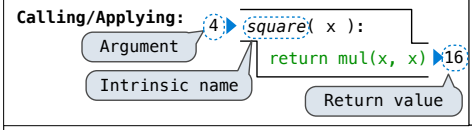
Defining:

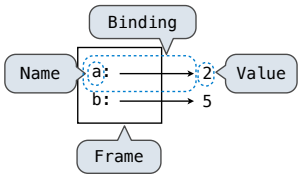


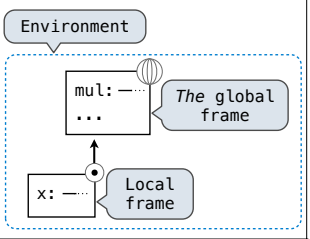
Call expression:

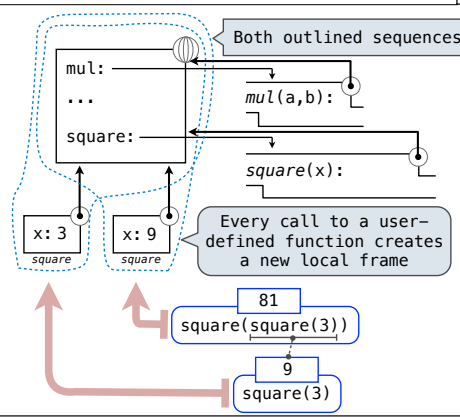


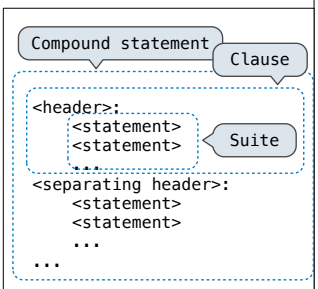
Calling/Applying:

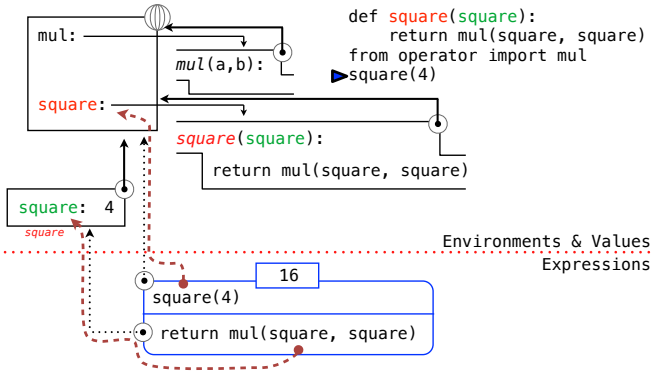


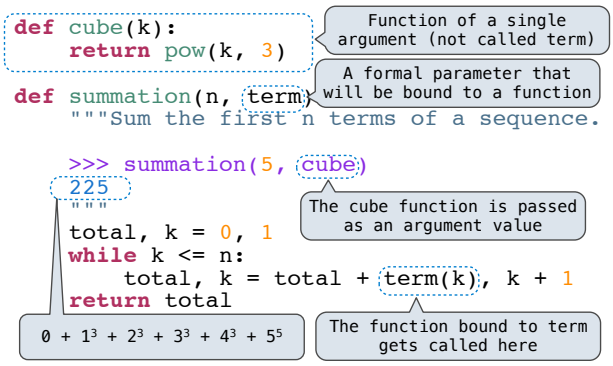












Higher-order function: A function that takes a function as an argument value or returns a function as a return value

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

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

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)
```

```
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

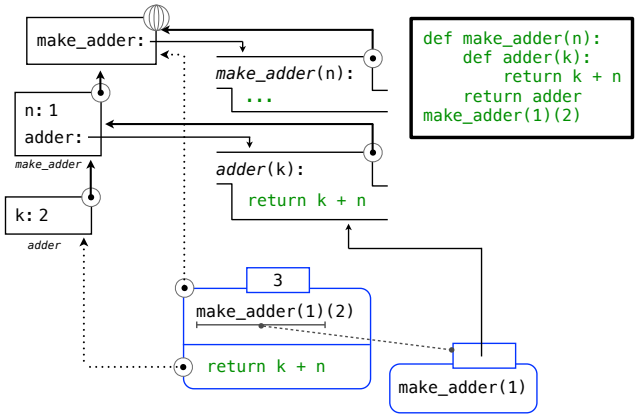
A function that returns a function

```
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
```

The name add_three is bound to a function

A local def statement

Can refer to names in the enclosing function

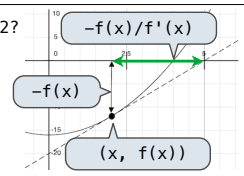



```
def make_adder(n):
    def adder(k):
        return k + n
    return adder
make_adder(1)(2)
```

- Compound objects combine primitive 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***

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

    guess -- An initial guess
    update -- A function from guesses to guesses; updates the guess
    done -- A function from guesses to boolean values; tests if guess is good

    >>> iter_improve(golden_update, golden_test)
    1.6180339887498895
    """
    k = 0
    while not done(guess) and k < max_updates:
        guess = update(guess)
        k = k + 1
    return guess
```

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

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

What does sum_squares need to know about square?

- Square takes one argument. **Yes**
- Square has the intrinsic name square. **No**
- Square computes the square of a number. **Yes**
- Square computes the square by calling mul. **No**

Rational numbers in the problem domain

add_rat mul_rat eq_rat

Rational numbers as numerators & denominators

make_rat numer denom

Rational numbers as tuples

tuple getitem

However tuples are implemented in Python

```
def mul_rat(x, y):
    """Multiply rational numbers x and y."""
    return make_rat(numer(x) * numer(y), denom(x) * denom(y))

def add_rat(x, y):
    """Add rational numbers x and y."""
    nx, dx = numer(x), denom(x)
    ny, dy = numer(y), denom(y)
    return make_rat(nx * dy + ny * dx, dx * dy)

def eq_rat(x, y):
    """Return whether rational numbers x and y are equal."""
    return numer(x) * denom(y) == numer(y) * denom(x)
```

```
def make_rat(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)
```

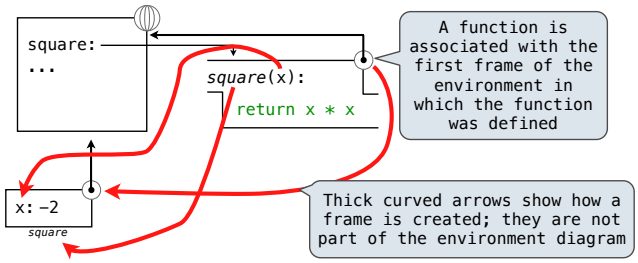
Three numeric types in Python:

```
>>> type(2)
<class 'int'>
>>> type(1.5)
<class 'float'>
>>> type(1+1j)
<class 'complex'>
```

Represents integers exactly

Represents real numbers approximately

```
def make_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)
```



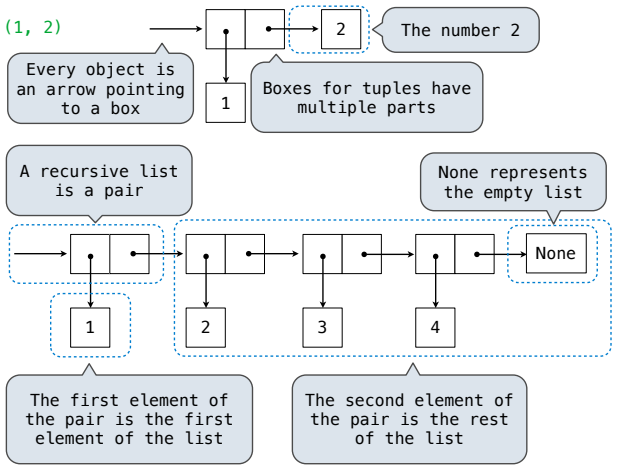
A function is associated with the first frame of the environment in which the function was defined

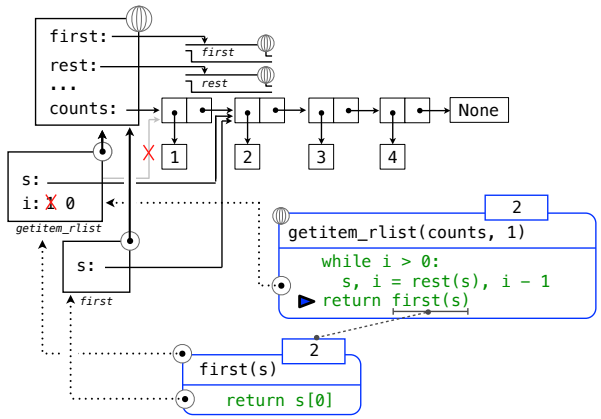
Thick curved arrows show how a frame is created; they are not part of the environment diagram

```
from operator import floordiv, mod

def divide_exact(n, d):
    """Return the quotient and remainder of dividing n by d.

    >>> q, r = divide_exact(13, 5)
    >>> q
    2
    >>> r
    3
    """
    return floordiv(n, d), mod(n, d)
```

Multiple assignment to two names

Multiple return values, separated by commas






```
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, ...

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

range(−2, 2)

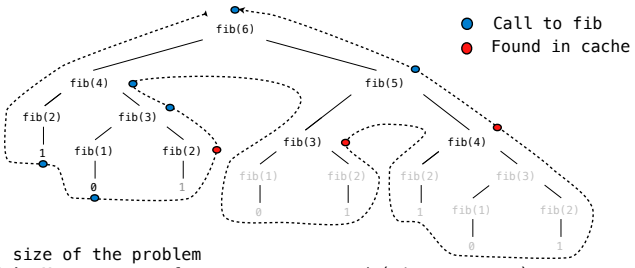
An element of a string is itself a string!

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.

(<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.



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

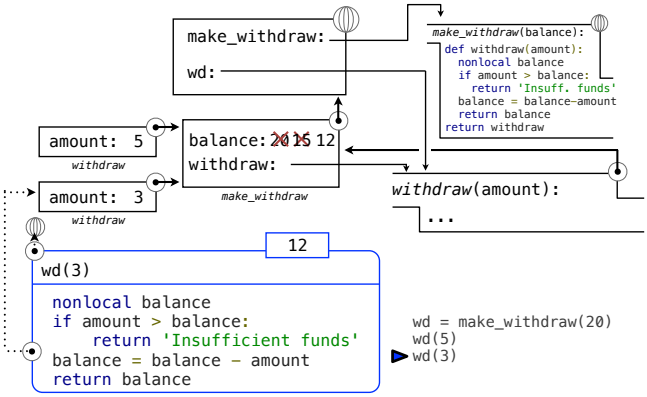
$$k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n)$$

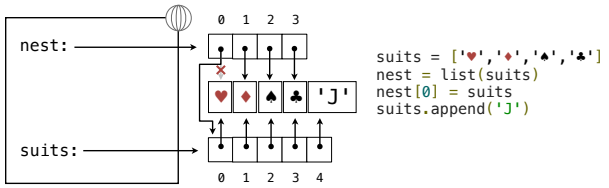
for sufficiently large values of ***n***.

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

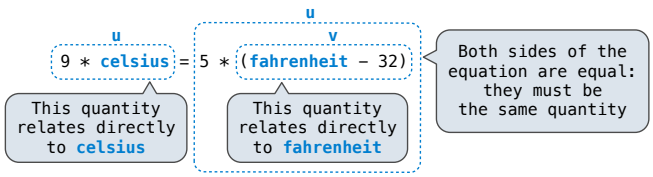
x = 2

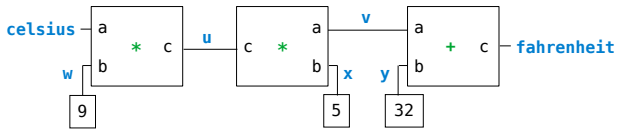
Status	Effect
•No nonlocal statement •"x" **is not** bound locally	Create a new binding from name "x" to object 2 in the first frame of the current environment.
•No nonlocal statement •"x" **is** bound locally	Re-bind name "x" to object 2 in the first frame of the current env.
•nonlocal x •"x" **is** 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





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.






```
def make_converter(c, f):
    u, v, w, x, y = [make_connector() for _ in range(5)]
    multiplier(c, w, u)
    multiplier(v, x, u)
    adder(v, y, f)
    constant(w, 9)
    constant(x, 5)
    constant(y, 32)
```

Connectors | Relations

```
def make_ternary_constraint(a, b, c, ab, ca, cb):
    """The constraint that ab(a,b)=c and ca(c,a)=b and cb(c,b)=a."""
    def new_value():
        av, bv, cv = [connector['has_val']() for connector in (a, b, c)]
        if av and bv:
            c['set_val'](constraint, ab(a['val'], b['val']))
        elif av and cv:
            b['set_val'](constraint, ca(c['val'], a['val']))
        elif bv and cv:
            a['set_val'](constraint, cb(c['val'], b['val']))
    def forget_value():
        for connector in (a, b, c):
            connector['forget'](constraint)
    constraint = {'new_val': new_value, 'forget': forget_value}
    for connector in (a, b, c):
        connector['connect'](constraint)
    return constraint


from operator import add, sub, mul, truediv

def adder(a, b, c):
    """The constraint that a + b = c."""
    return make_ternary_constraint(a, b, c, add, sub, sub)

def multiplier(a, b, c):
    """The constraint that a * b = c."""
    return make_ternary_constraint(a, b, c, mul, truediv, truediv)
```

```
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.
- As soon as an instance is created, it is passed to __init__, which is an attribute 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.

```
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
```

```
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!
```

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

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

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

```
class 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))
```

- Objects have **local state** & interact via **message passing**
- Objects are **instantiated** by classes, which are also objects
- Classes may **inherit** from other classes to share behavior
- Mechanics of objects are governed by **"evaluation procedures"**

THE LINE

- Objects have **mutable dictionaries** of attributes
- **Attribute look-up for instances** is a function
- **Attribute look-up for classes** is another function
- Object **instantiation** is another function

```
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:
        init(instance, *args)
    return instance
def make_account_class():
    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')
    def withdraw(self, amount):
        ...
    return make_class({'__init__': __init__,'deposit':  deposit,
                       'withdraw': withdraw,'interest': 0.02})
```

```
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)

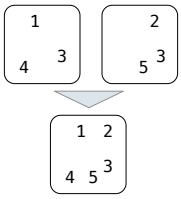
1. Attempt to coerce arguments into values of the same type
2. Apply type-specific (not cross-type) operations

```
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)
```

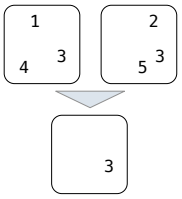
CS 61A Final Exam Study Guide – Page 1

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.

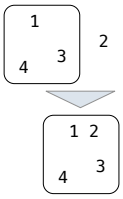
Union



Intersection



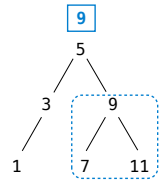
Adjunction



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



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

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

Exceptions are raised with a raise statement.

```
raise <expression>
```

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

Exceptions are constructed like any other object; they are just instances of classes that inherit from BaseException.

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

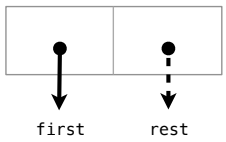
The <try suite> is executed first;

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

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

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

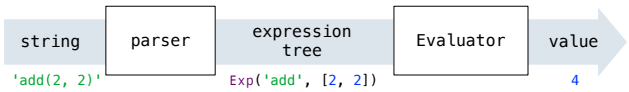
Streams are lazily computed recursive lists



Stored explicitly *Evaluated lazily*

```
class Stream(object):
    def __init__(self, first, compute_rest, empty=False):
        self.first = first
        self._compute_rest = compute_rest
        self.empty = empty
        self._rest = None
        self._computed = False

    @property
    def rest(self):
        assert not self.empty, 'Empty streams have no rest.'
        if not self._computed:
            self._rest = self._compute_rest()
            self._computed = True
        return self._rest

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

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

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

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



An *expression tree* is a (hierarchical) data structure that represents a (nested) expression.

```
class Exp(object):
    """A call expression in Calculator."""
    def __init__(self, operator, operands):
        self.operator = operator
        self.operands = operands
```

```
def calc_parse(line):
    """Parse a line of calculator input."""
    tokens = tokenize(line)
    expression_tree = analyze(tokens)
```

Lexical analysis is also called **tokenization**

Lexical analyzer: Analyzes an input string as a sequence of tokens, which are symbols and delimiters.

```
>>> tokenize('add(2, mul(4, 6))')
['add', '(', '2', ',', 'mul', '(', '4', ',', '6', ')', ')']
```

Symbol: a built-in operator name | Delimiter | Symbol: a literal | Delimiter

Syntactic analyzer: Analyzes a sequence of tokens as an expression tree, which typically includes call expressions.

```
['add','(','2',',','3',')']
    ['(','2',',','3',')']
        ['2',',','3',')']
            ['2',',','3',')']
```

Pass 1	Pass 2
[',','3',')']	['3',')']
	[')']
	[]

```
def analyze(tokens):
    token = analyze_token(tokens.pop(0))
    if type(token) in (int, float):
        return token
    else:
        tokens.pop(0)  # Remove (
        return Exp(token, analyze_operands(tokens))

def analyze_operands(tokens):
    operands = []
    while tokens[0] != ')':
        if operands:
            tokens.pop(0)  # Remove ,
        operands.append(analyze(tokens))
    tokens.pop(0)  # Remove )
    return operands
```

```
def calc_eval(exp):
    """Evaluate a Calculator expression."""
    if type(exp) in (int, float):
        return exp
    elif type(exp) == Exp:
        arguments = list(map(calc_eval, exp.operands))
        return calc_apply(exp.operator, arguments)
```

Numbers are self-evaluating

```
def calc_apply(operator, args):
    """Apply the named operator to a list of args."""
    if operator in ('add', '+'):
        return sum(args)
    ...
```

Dispatch on operator name

Implement operator logic in Python

```
def read_eval_print_loop():
    """Run a read-eval-print loop for calculator."""
    while True:
        expression_tree = calc_parse(input('calc> '))
        print(calc_eval(expression_tree))
```

```
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)
```

```
>>> letters = Letters()
>>> letters.__next__()
'a'
>>> letters.__next__()
'b'
>>> letters.__next__()
'c'
>>> letters.__next__()
'd'
>>> letters.__next__()
Traceback ...
StopIteration

>>> for item in Letters():
        print(item)
a
b
c
d

>>> i = Letters().__iter__()
>>> try:
        while True:
            item = i.__next__()
            print(item)
    except StopIteration:
        pass
a
b
c
d
```

Words are strings without spaces, representing text, numbers, and boolean values.

Sentences are immutable sequences of words and sentences

```
? print "hello          ? print [hello world]
hello                   hello world
? print "sum            ? show [hello world]
sum                     [hello world]
? print "2
2
```

Sentences can be constructed from words or sentences

sentence	Output a sentence containing all elements of two sentences. Input words are converted to sentences.
list	Output a sentence containing the two inputs.
fput	Output a sentence containing the first input and all elements in the second input.

The run procedure evaluates a sentence as a line of Logo code and outputs its value:

```
? run [print sum 1 2]
3
? print run sentence "sum sentence 10 run [difference 7 3]
14
```

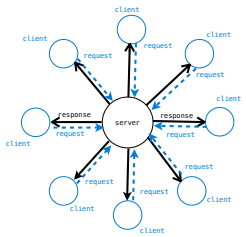
14 [10 4] 4

Procedure definition is a special form, not a call expression

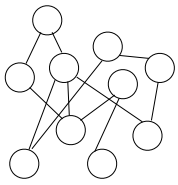
Procedure name | Formal parameter | Body

```
? to double :x
> output sum :x :x
> end
? print double 4
8
```

Client-server vs **Peer-to-peer**

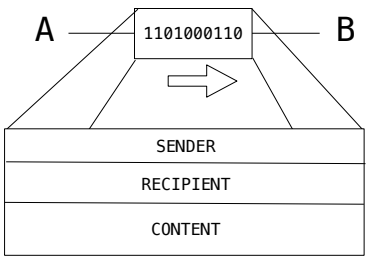


Appropriate for dispensing a service
Clients make requests from server
Server listens for requests and responds to them.
Many clients, but only 1 server.



Division of labor among *all* computers
Applications:
- Data storage
- Communication
- Large-scale computation

All computers send and receive data.
All computer contribute resources:
- Disk space
- Memory
- Processing power

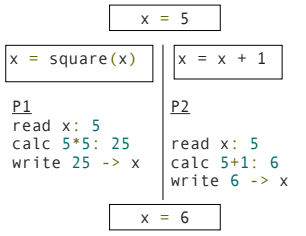


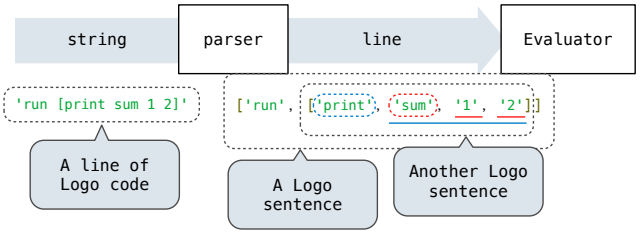
Anatomy of a network message

Where to send a response

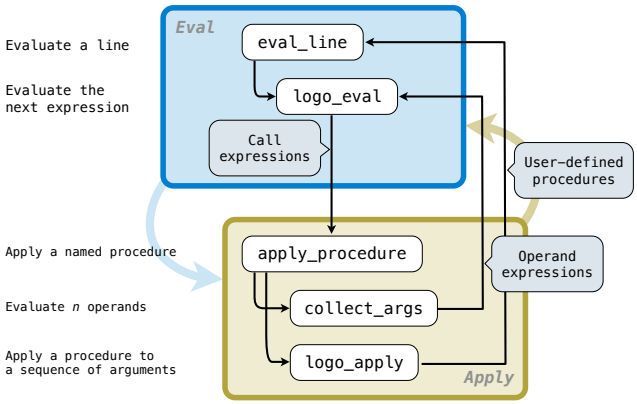
Where to route message

Data, signal, encoded video, text, etc.





The logo_eval function dispatches on expression form:
- A **primitive expression** is a word that can be interpreted as a number, True, or False. Primitives are self evaluating.
- A **variable** is looked up in the current environment.
- A **procedure definition** creates a new user-defined procedure.
- A **quoted expression** evaluates to the text of the quotation, which is a string without the preceding quote. Sentences are quoted and evaluate to themselves.
- A **call expression** is evaluated with apply_procedure.

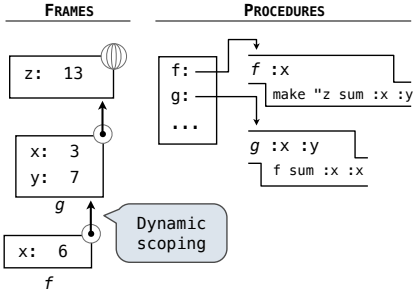


Logo binds variable names to values, as does Python.
An environment stores name bindings in a sequence of frames.
Each frame can have at most one value bound to a given name.
The make procedure adds or changes variable bindings:

```
? make "x 2
```

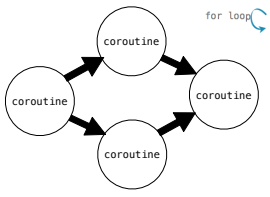
Values bound to names are looked up using variable expressions:

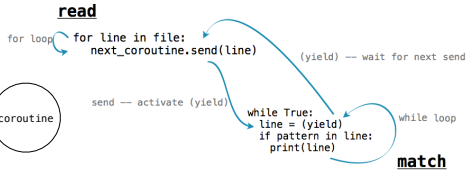
```
? print :x
2
```




```
? to f :x
> make "z sum :x :y
> end
? to g :x :y
> f sum :x :x
> end
? g 3 7
? print :z
13
```

read





match

```
>>> text = 'dig dog bog.'
>>> matcher = match('og')
>>> matcher.__next__()
Looking for og
>>> read(text, matcher)
dog
bog.
=== Done ===
```

```
def read(text, next_coroutine):
    for line in text.split():
        next_coroutine.send(line)
    next_coroutine.close()

def match(pattern):
    print('Looking for ' + pattern)
    try:
        while True:
            s = (yield)
            if pattern in s:
                print(s)
    except GeneratorExit:
        print("=== Done ===")
```

```
def make_withdraw(balance)
        balance_lock = Lock()
        def withdraw(amount):
            nonlocal balance
            # try to acquire the lock
            balance_lock.acquire()
            # once successful, enter the critical section
            if amount > balance:
                print("Insufficient funds")
            else:
                balance = balance - amount
                print(balance)
            # upon exiting the critical section, release the lock
            balance_lock.release()
```