CS 61A Structure and Interpretation of Computer Programs

Fall 2014 Final Exam

INSTRUCTIONS

- You have 3 hours to complete the exam.
- The exam is closed book, closed notes, closed computer, closed calculator, except one hand-written 8.5” × 11” crib sheet of your own creation and the 3 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	Q. 6	Total
/14	/16	/12	/12	/18	/8	/80

Blank Page

1. (14 points) Representing Scheme Lists

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`.

The `Pair` class from Project 4 is described on your final study guide. Recall that its `__str__` method returns a Scheme expression, and its `__repr__` method returns a Python expression. The full implementation of `Pair` and `nil` appear at the end of the exam as an appendix. Assume that you have started Python 3, loaded `Pair` and `nil` from `scheme_reader.py`, then executed the following:

```
blue = Pair(3, Pair(4, nil))
gold = Pair(Pair(6, 7), Pair(8, 9))

def process(s):
    cal = s
    while isinstance(cal, Pair):
        cal.bear = s
        cal = cal.second
    if cal is s:
        return cal
    else:
        return Pair(cal, Pair(s.first, process(s.second)))

def display(f, s):
    if isinstance(s, Pair):
        print(s.first, f(f, s.second))

y = lambda f: lambda x: f(f, x)
```

Expression	Output
`Pair(1, nil)`	`Pair(1, nil)`
`print(Pair(1, nil))`	`(1)`
`1/0`	ERROR
`print(print(3), 1/0)`	
`print(Pair(2, blue))`	
`print(gold)`	

Expression	Output
`process(blue.second)`	
`print(process(gold))`	
`gold.second.bear.first`	
`y(display)(gold)`	

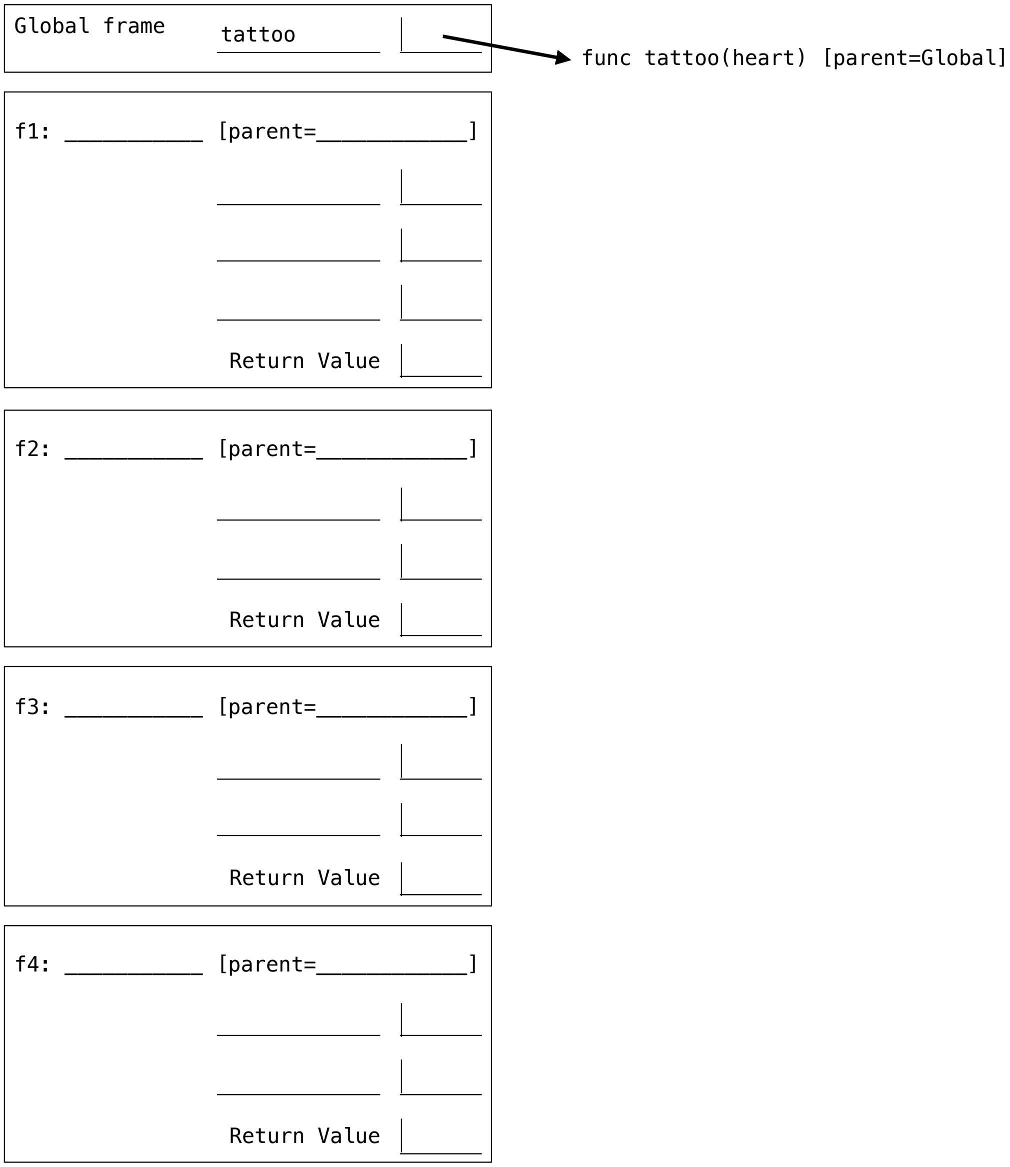
2. (16 points) Environments

(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.

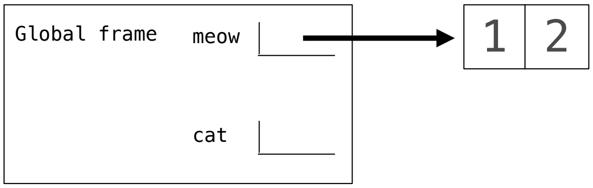
```
def tattoo(heart):
    def mom():
        nonlocal mom
        mom = lambda: heart(2) + 1
        return 3
    return mom() + mom() + 4

tattoo(lambda ink: ink + 0.5)
```


Global frame
tattoo
func tattoo(heart) [parent=Global]
f1: ___________ [parent=____________]
Return Value
f2: ___________ [parent=____________]
Return Value
f3: ___________ [parent=____________]
Return Value
f4: ___________ [parent=____________]
Return Value

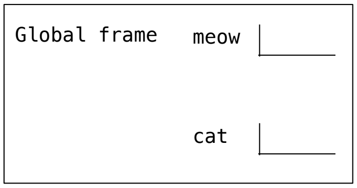
(b) **(6 pt)** For the six-line program below, fill in the three environment diagrams that would result after executing each pair of lines in order. **You must use box-and-pointer diagrams to represent list values. You do not need to write the word "list" or write index numbers.**

Important: All six lines of code are executed in order! Line 3 is executed after line 2 and line 5 after line 4.

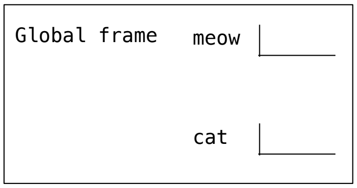
```
1 meow = [1, 2]
2 cat =  [meow, [4, 5]]
```



```
3 cat[0][1] = cat[1][0]
4 cat[meow[0]][0] = meow
```



```
5 meow[0] = [cat.pop(0)]
6 cat.extend(cat[0][1:])
```



(c) **(2 pt)** Circle the value, `True` or `False`, of each expression below when evaluated in the environment created by executing all six lines above. If you leave this question blank, you will receive 1 point.

Circle *True* or *False*: `meow is cat[0]`

Circle *True* or *False*: `meow[0][0] is cat[0][0]`

3. (12 points) Expression Trees

Your partner has created an interpreter for a language that can add or multiply positive integers. Expressions are represented as instances of the `Tree` class and must have one of the following three forms:

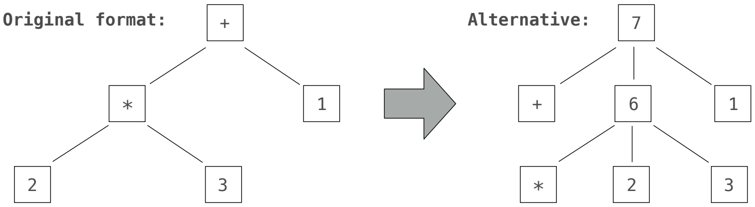
- (**Primitive**) A positive integer `entry` and no branches, representing an integer
- (**Combination**) The `entry '+'`, representing the sum of the values of its branches
- (**Combination**) The `entry '*'`, representing the product of the values of its branches

The `Tree` class is on the Midterm 2 Study Guide. The sum of no values is 0. The product of no values is 1.

(a) (6 pt) Unfortunately, multiplication in Python is broken on your computer. Implement `eval_with_add`, which evaluates an expression without using multiplication. You may fill the blanks with names or call expressions, but the only way you are allowed to combine two numbers is using addition.

```
def eval_with_add(t):
    """Evaluate an expression tree of * and + using only addition.

    >>> plus = Tree('+', [Tree(2), Tree(3)])
    >>> eval_with_add(plus)
    5
    >>> times = Tree('*', [Tree(2), Tree(3)])
    >>> eval_with_add(times)
    6
    >>> deep = Tree('*', [Tree(2), plus, times])
    >>> eval_with_add(deep)
    60
    >>> eval_with_add(Tree('*'))
    1
    """
    if t.entry == '+':

        return sum(________________________________________________________)

    elif t.entry == '*':


        total = __________________________________________________________


        for b in t.branches:


            total, term = 0, ____________________________________________


            for __________ in ___________________________________________:


                total = total + term

        return total

    else:

        return t.entry
```

(b) **(6 pt)** A TA suggests an alternative representation of an expression, in which the `entry` is the value of the expression. For combinations, the operator appears in the left-most (index 0) branch as a leaf.



Implement `transform`, which takes an expression and mutates all combinations so that their entries are values and their first branches are operators. In addition, `transform` should return the value of its argument. You may use the `calc_apply` function defined below.

```
def calc_apply(operator, args):
    if operator == ’+’:
        return sum(args)
    elif operator == ’*’:
        return product(args)

def product(vals):
    total = 1
    for v in vals:
        total *= v
    return total

def transform(t):
    """Transform expression tree t to have value entries and operator leaves.

    >>> seven = Tree(’+’, [Tree(’*’, [Tree(2), Tree(3)]), Tree(1)])
    >>> transform(seven)
    7
    >>> seven
    Tree(7, [Tree(+), Tree(6, [Tree(*), Tree(2), Tree(3)]), Tree(1)])
    """

    if t.branches:

        args = []

        for b in t.branches:


            args.append(__________________________________________________)


        t.branches = __________________________________________________


        t.entry = __________________________________________________


    return __________________________________________________
```

4. (12 points) Lazy Sunday

(a) **(4 pt)** A *flat-map* operation maps a function over a sequence and flattens the result. Implement the `flat_map` method of the `FlatMapper` class. You may use at most 3 lines of code, indented however you choose.

```
class FlatMapper:

    """A FlatMapper takes a function fn that returns an iterable value. The
    flat_map method takes an iterable s and returns a generator over all values
    in the iterables returned by calling fn on each element of s.

    >>> stutter = lambda x: [x, x]
    >>> m = FlatMapper(stutter)
    >>> g = m.flat_map((2, 3, 4, 5))
    >>> type(g)
    <class 'generator'>
    >>> list(g)
    [2, 2, 3, 3, 4, 4, 5, 5]
    """

    def __init__(self, fn):

        self.fn = fn

    def flat_map(self, s):

        ________________________________________________________________________

        ________________________________________________________________________

        ________________________________________________________________________
```

(b) **(2 pt)** Define `cycle` that returns a `Stream` repeating the digits 1, 3, 0, 2, and 4. **Hint**: `(3+2)%5` equals 0.

```
def cycle(start=1):
    """Return a stream repeating 1, 3, 0, 2, 4 forever.

    >>> first_k(cycle(), 12)  # Return the first 12 elements as a list
    [1, 3, 0, 2, 4, 1, 3, 0, 2, 4, 1, 3]
    """

    def compute_rest():

        return ____________________________________________________________

    return Stream(_____________________________, _________________________)
```

(c) (4 pt) Implement the Scheme procedure `directions`, which takes a number `n` and a symbol `sym` that is bound to a nested list of numbers. It returns a Scheme expression that evaluates to `n` by repeatedly applying `car` and `cdr` to the nested list. Assume that `n` appears exactly once in the nested list bound to `sym`.

Hint: The implementation searches for the number `n` in the nested list `s` that is bound to `sym`. The returned expression is built during the search. See the tests at the bottom of the page for usage examples.

```
(define (directions n sym)

    (define (search s exp)

        ; Search an expression s for n and return an expression based on exp.

        (cond ((number? s) _____________________________________________________)

              ((null? s) nil)

              (else (search-list s exp))))

    (define (search-list s exp)

        ; Search a nested list s for n and return an expression based on exp.

        (let ((first ___________________________________________________________)

              (rest  ___________________________________________________________))

             (if (null? first) rest first)))

    (search (eval sym) sym))

(define a '(1 (2 3) ((4))))
(directions 1 'a)
; expect (car a)
(directions 2 'a)
; expect (car (car (cdr a)))
(define b '((3 4) 5))
(directions 4 'b)
; expect (car (cdr (car b)))
```

(d) (2 pt) What expression will `(directions 4 'a)` evaluate to?

____

5. (18 points) Basis Loaded

Ben Bitdiddle notices that any positive integer can be expressed as a sum of powers of 2. Some examples:

$$\begin{aligned} 11 &= 8 + 2 + 1 \\ 23 &= 16 + 4 + 2 + 1 \\ 24 &= 16 + 8 \\ 45 &= 32 + 8 + 4 + 1 \\ 2014 &= 1024 + 512 + 256 + 128 + 64 + 16 + 8 + 4 + 2 \end{aligned}$$

A `basis` is a linked list of decreasing integers (such as powers of 2) with the property that any positive integer `n` can be expressed as the sum of elements in the `basis`, starting with the largest element that is less than or equal to `n`.

(a) **(4 pt)** Implement `sum_to`, which takes a positive integer `n` and a linked list of decreasing integers `basis`. It returns a linked list of elements of the `basis` that sum to `n`, starting with the largest element of `basis` that is less than or equal to `n`. If no such sum exists, raise an `ArithmeticError`. **Each number in `basis` can only be used once (or not at all).** The `Link` class is described on your Midterm 2 Study Guide.

```
def sum_to(n, basis):
    """Return elements of linked list basis that sum to n.

    >>> twos = Link(32, Link(16, Link(8, Link(4, Link(2, Link(1))))))
    >>> sum_to(11, twos)
    Link(8, Link(2, Link(1)))
    >>> sum_to(23, twos)
    Link(16, Link(4, Link(2, Link(1))))
    >>> sum_to(24, twos)
    Link(16, Link(8))
    >>> sum_to(45, twos)
    Link(32, Link(8, Link(4, Link(1))))
    """


    if ______________________________________________________________________:


        return Link.empty


    elif ____________________________________________________________________:


        raise ArithmeticError


    elif basis.first > n:


        return sum_to(n, basis.rest)


    else:


        return ______________________________________________________________
```

(b) (6 pt) Cross out as many lines as possible in the implementation of the FibLink class so that all doctests pass. A FibLink is a subclass of Link that contains decreasing Fibonacci numbers. The up_to method returns a FibLink instance whose first element is the largest Fibonacci number that is less than or equal to positive integer n.

```
class FibLink(Link):
    """Linked list of Fibonacci numbers.

    >>> ten = FibLink(2, FibLink(1)).up_to(10)
    >>> ten
    Link(8, Link(5, Link(3, Link(2, Link(1)))))
    >>> ten.up_to(1)
    Link(1)
    >>> six, thirteen = ten.up_to(6), ten.up_to(13)
    >>> six
    Link(5, Link(3, Link(2, Link(1))))
    >>> thirteen
    Link(13, Link(8, Link(5, Link(3, Link(2, Link(1))))))
    """
    successor = self.first + self.rest
    @property
    def successor():
    def successor(self):
        return first + rest.first
        return self.first + self.rest.first

    def up_to(n):
    def up_to(self, n):
        while self.first > n:
            self = self.rest.first
            self = rest
            self.first = self.rest.first
        if self.first == n:
            return self
        elif self.first > n:
            return self.up_to(n)
            return self.rest.up_to(n)
        elif self.successor > n:
        elif self.first < n:
            return self
        else:
            return FibLink(self.successor(self), self).up_to(n)
            return FibLink(self.successor, self).up_to(n)
            return FibLink(self.successor(self), self.rest).up_to(n)
            return FibLink(self.successor, self.rest).up_to(n)
```

(c) (2 pt) Circle the Θ expression below that describes the number of calls made to FibLink.up_to when evaluating FibLink(2, FibLink(1)).up_to(n). The constant ϕ is $\frac{1+\sqrt{5}}{2} = 1.618...$

$\Theta(1)$ $\Theta(\log_\phi n)$ $\Theta(n)$ $\Theta(n^2)$ $\Theta(\phi^n)$

(d) **(2 pt)** Alyssa P. Hacker remarks that Fibonacci numbers also form a basis. How many **total** calls to `FibLink.up_to` will be made while evaluating **all** the doctests of the `fib_basis` function below? Assume that `sum_to` and `FibLink` are implemented correctly. Write your answer in the box.

```
def fib_basis():
    """Fibonacci basis with caching.

    >>> r = fib_basis()
    >>> r(11)
    Link(8, Link(3))
    >>> r(23)
    Link(21, Link(2))
    >>> r(24)
    Link(21, Link(3))
    >>> r(45)
    Link(34, Link(8, Link(3)))
    """
    fibs = FibLink(2, FibLink(1))
    def represent(n):
        nonlocal fibs
        fibs = fibs.up_to(n)
        return sum_to(n, fibs)
    return represent
```

(e) **(4 pt)** Implement `fib_sums`, a function that takes positive integer `n` and returns the number of ways that `n` can be expressed as a sum of unique Fibonacci numbers. Assume that `FibLink` is implemented correctly.

```
def fib_sums(n):
    """The number of ways n can be expressed as a sum of unique Fibonacci numbers.

    >>> fib_sums(9)  # 8+1, 5+3+1
    2
    >>> fib_sums(12) # 8+3+1
    1
    >>> fib_sums(13) # 13, 8+5, 8+3+2
    3
    """
    def sums(n, fibs):
        """Ways n can be expressed as a sum of elements in fibs."""

        if n == 0:

            return 1

        elif _________________________________________________________________:

            return 0

        a = __________________________________________________________________


        b = __________________________________________________________________

        return a + b

    return sums(n, FibLink(2, FibLink(1)).up_to(n))
```

6. (8 points) Sequels

Assume that the following table of movie ratings has been created.

```
create table ratings as
  select "The Matrix" as title,     9 as rating union
  select "The Matrix Reloaded",     7           union
  select "The Matrix Revolutions",  5           union
  select "Toy Story",               8           union
  select "Toy Story 2",             8           union
  select "Toy Story 3",             9           union
  select "Terminator",              8           union
  select "Judgment Day",            9           union
  select "Rise of the Machines",    5;
```

Correct output
Judgment Day
Terminator
The Matrix
Toy Story
Toy Story 2
Toy Story 3

The correct output table for both questions below happens to be the same. It appears above to the right for your reference. **Do not hard code your solution to work only with this table!** Your implementations should work correctly even if the contents of the `ratings` table were to change.

(a) (2 pt) Select the titles of all movies that have a rating greater than 7 in alphabetical order.

(b) (6 pt) Select the titles of all movies for which at least 2 other movies have the same rating. The results should appear in alphabetical order. Repeated results are acceptable. *You may only use the SQL features introduced in this course.*

```
with

  groups(name, score, n) as (

    select _____________, ________________, ___________ from ratings union

    select _____________, ________________, ________ from groups, ratings

      where ______________________________________________________________

  )

select title from ________________________________________________________

  where __________________________________________________________________

  order by ______________________________________________________________;
```

Appendix: Pair and nil Implementations

This page does not contain a question. These classes were originally defined in `scheme_reader.py`.

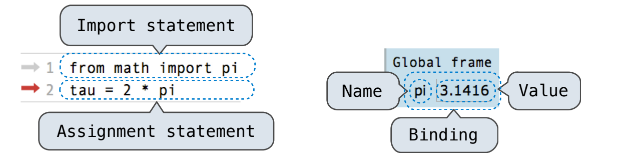
```
class Pair:
    """A pair has two instance attributes: first and second.  For a Pair to be
    a well-formed list, second is either a well-formed list or nil.  Some
    methods only apply to well-formed lists.

    >>> s = Pair(1, Pair(2, nil))
    >>> s
    Pair(1, Pair(2, nil))
    >>> print(s)
    (1 2)
    """
    def __init__(self, first, second):
        self.first = first
        self.second = second

    def __repr__(self):
        return "Pair({0}, {1})".format(repr(self.first), repr(self.second))

    def __str__(self):
        s = "(" + str(self.first)
        second = self.second
        while isinstance(second, Pair):
            s += " " + str(second.first)
            second = second.second
        if second is not nil:
            s += " . " + str(second)
        return s + ")"

class nil:
    """The empty list"""

    def __repr__(self):
        return "nil"

    def __str__(self):
        return "()"

    def __len__(self):
        return 0

    def __getitem__(self, k):
        if k < 0:
            raise IndexError("negative index into list")
        raise IndexError("list index out of bounds")

    def map(self, fn):
        return self

nil = nil() # Assignment hides the nil class; there is only one instance
```

Scratch Paper

Scratch Paper

Scratch Paper

Scratch Paper



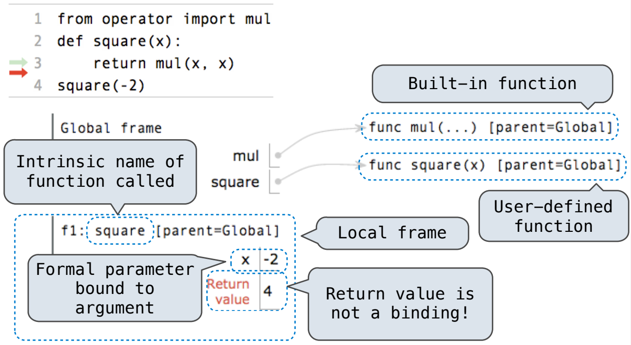
Code (left):

Statements and expressions
Red arrow points to next line.
Gray arrow points to the line just executed

Frames (right):

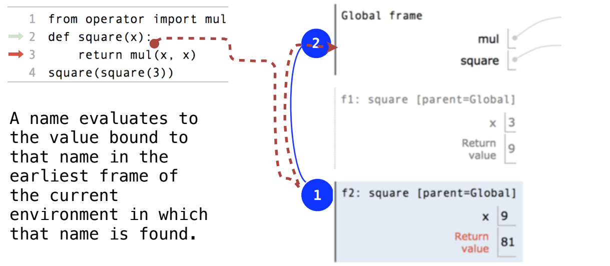
A name is bound to a value

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



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

Built-in function
Global frame
Intrinsic name of function called
mul
square
func mul(...) [parent=Global]
func square(x) [parent=Global]
User-defined function
f1: square [parent=Global]
Local frame
Formal parameter bound to argument
x -2
Return value 4
Return value is not a binding!



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

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:

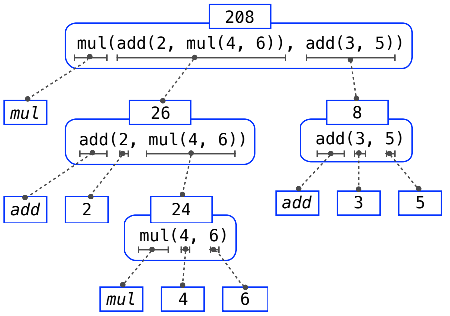
1. Evaluate the subexpression <left>.
2. If the result is a false value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

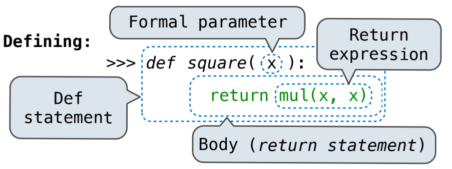
Evaluation rule for not expressions:

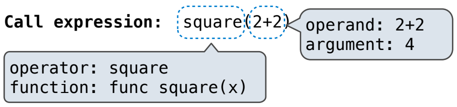
1. Evaluate <exp>; The value is True if the result is a false value, and False otherwise.

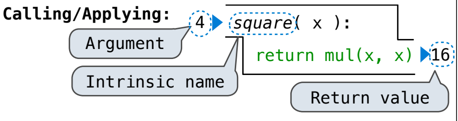
Execution rule for while statements:

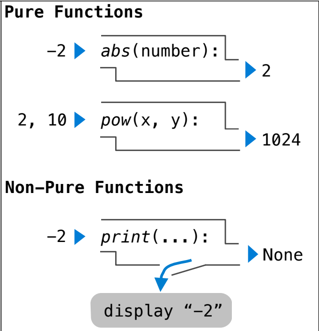
1. Evaluate the header's expression.
2. If it is a true value, execute the (*whole*) suite, then return to step 1.

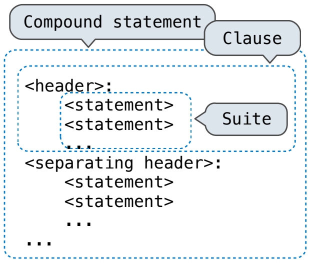


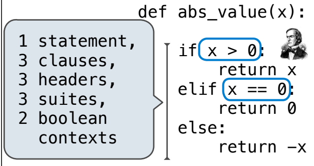




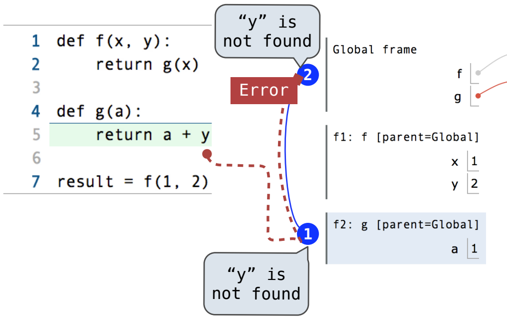






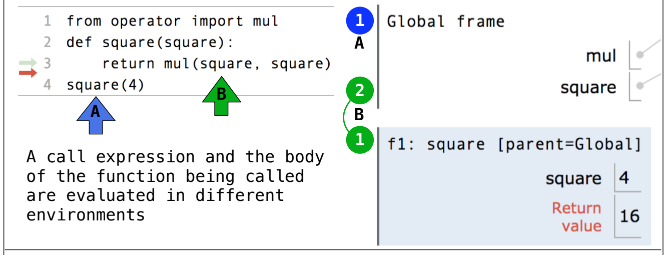


```
def abs_value(x):
    if x > 0:
        return x
    elif x == 0:
        return 0
    else:
        return -x
```



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

def g(a):
    return a + y

result = f(1, 2)
```

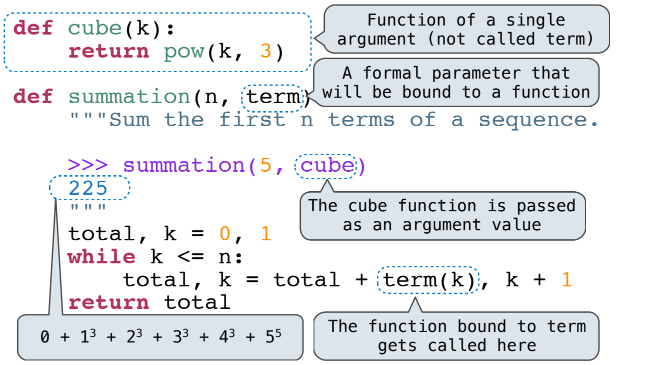
- An environment is a sequence of frames
- An environment for a non-nested function (no def within def) consists of one local frame, followed by the global frame



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

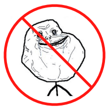
A call expression and the body of the function being called are evaluated in different environments

```
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
$0 + 1^3 + 2^3 + 3^3 + 4^3 + 5^5$
The function bound to term gets called here



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

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

```
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

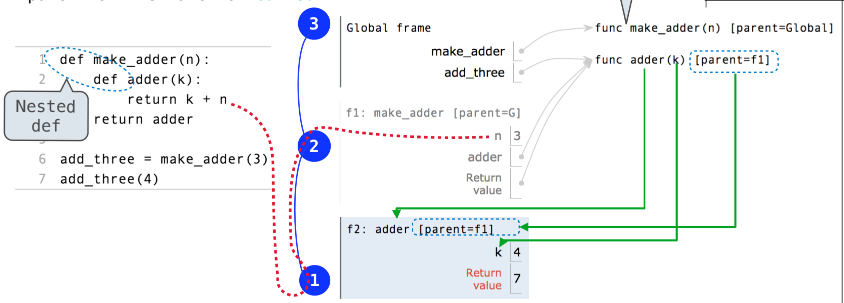
The name add_three is bound to a function

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

add_three = make_adder(3)
add_three(4)
```

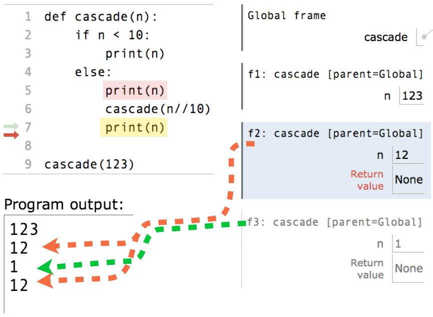
```
def curry2(f):
    """Returns a function g such that g(x)(y) returns f(x, y)."""
    def g(x):
        def h(y):
            return f(x, y)
        return h
    return g
```

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
```



```
def cascade(n):
    if n < 10:
        print(n)
    else:
        print(n)
        cascade(n//10)
        print(n)

cascade(123)
```

Program output:

```
123
12
1
12
```

Global frame
cascade
func cascade(n) [parent=Global]
f1: cascade [parent=Global]
n 123
f2: cascade [parent=Global]
n 12
Return value None
f3: cascade [parent=Global]
n 1
Return value None

- 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:

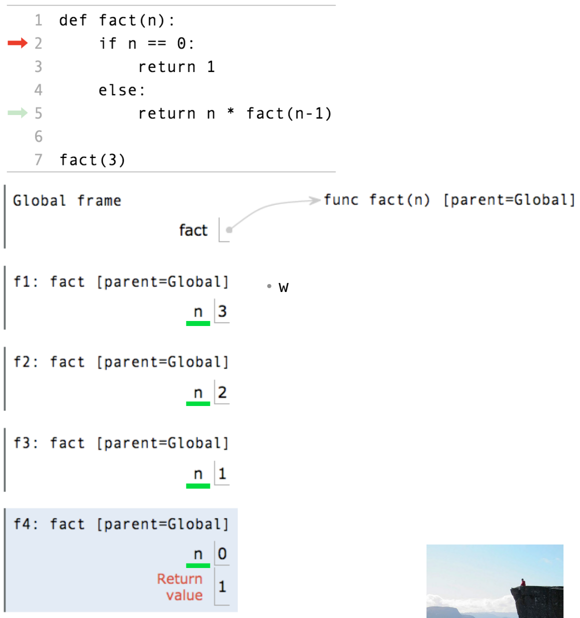
1. Create a **function value**: func *<name>*(*<formal parameters>*)
2. Its parent is the current frame.

   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:

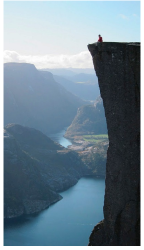
1. Add a **local frame**, titled with the *<name>* of the function being called.
2. Copy the parent of the function to the **local frame**: [parent=*<label>*]
3. Bind the *<formal parameters>* to the arguments in the **local frame**.
4. Execute the body of the function in the environment that starts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

CS 61A Midterm 2 Study Guide — Page 1

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:

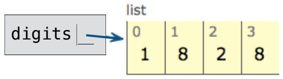
```
def pair(x, y):
    """Return a functional pair."""
    def get(index):
        if index == 0:
            return x
        elif index == 1:
            return y
    return get

def select(p, i):
    """Return element i of pair p."""
    return p(i)

>>> p = pair(1, 2)
>>> select(p, 0)
1
>>> select(p, 1)
2
```
This function represents a pair

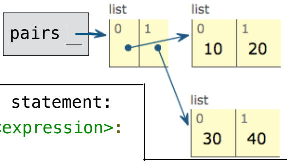
Constructor is a higher-order function

Selector defers to the object itself

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>

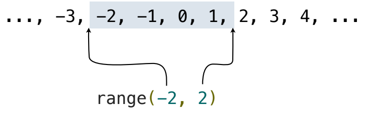
Unpacking in a for statement:

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
```



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

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

>>> list(range(4))
[0, 1, 2, 3]
```
List constructor

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

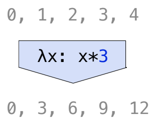
List comprehensions:

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

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

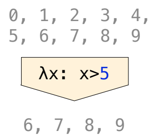
A combined expression that evaluates to a list using this evaluation procedure:

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

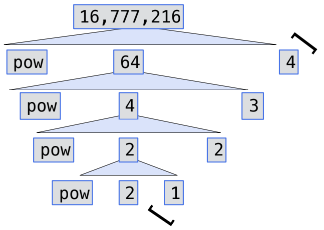
```
def apply_to_all(map_fn, s):
    """Apply map_fn to each element of s.

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



```
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)]
```



```
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 $k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n)$ for all n larger than some m

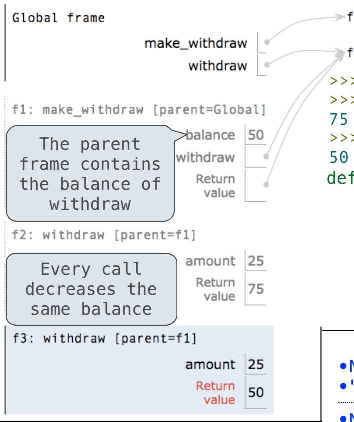
$\Theta(b^n)$ Exponential growth. Recursive fib takes $\Theta(\phi^n)$ steps, where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61828$
Incrementing the problem scales R(n) by a factor

$\Theta(n^2)$ 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


Global frame
make_withdraw
withdraw
func make_withdraw(balance) [parent=Global]
func withdraw(amount) [parent=f1]
f1: make_withdraw [parent=Global]
balance
50
withdraw
Return value
The parent frame contains the balance of withdraw
f2: withdraw [parent=f1]
amount
25
Return value
75
Every call decreases the same balance
f3: withdraw [parent=f1]
amount
25
Return value
50

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

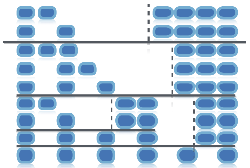
$\Theta(n^2)$ $\Theta(n^2 + n)$ $\Theta(n^2 + 500 \cdot n + \log_2 n + 1000)$

CS 61A Midterm 2 Study Guide — Page 2

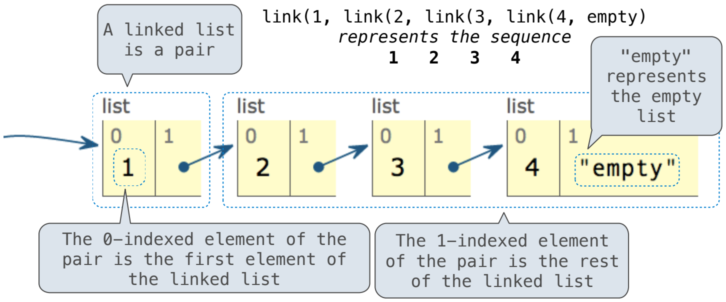
Linked list data abstraction:

```
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 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)
```



```
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)))
```


A linked list is a pair

link(1, link(2, link(3, link(4, empty)

represents the sequence

1 2 3 4

"empty" represents the empty list

The 0-indexed element of the pair is the first element of the linked list

The 1-indexed element of the pair is the rest of the linked list

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

Sequence abstraction special names:

__getitem__	Element selection []
__len__	Built-in len function

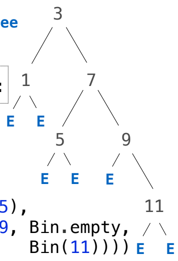
Yes, this call is recursive

```
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
```

When a class is called:
1. A new instance of that class is created:

An account instance

balance: 0   holder: 'Jim'

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

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

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

>>> 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
```

CS 61A Final Exam Study Guide – Page 1

Exceptions are raised with a raise statement.

```
raise <expression>
```

<expression> must evaluate to a subclass of BaseException or an instance of one.

Exceptions are constructed like any other object. E.g., TypeError('Bad argument!')

```
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.

```
>>> try:
        x = 1/0
    except ZeroDivisionError as e:
        print('handling a', type(e))
        x = 0

handling a <class 'ZeroDivisionError'>
>>> x
0
```

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

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

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

```
>>> counts = [1, 2, 3]
>>> for item in counts:
        print(item)

1
2
3
```

```
>>> items = counts.__iter__()
>>> try:
        while True:
            item = items.__next__()
            print(item)
    except StopIteration:
        pass
```

```
class FibIter:
    def __init__(self):
        self._next = 0
        self._addend = 1

    def __next__(self):
        result = self._next
        self._addend, self._next = self._next, self._addend + self._next
        return result
```

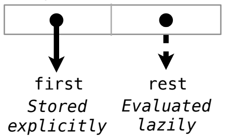
"Please don't reference these directly. They may change."

```
>>> fibs = FibIter()
>>> [next(fibs) for _ in range(10)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

A stream is a linked list, but the rest of the list is computed on demand.

Once created, Streams and Rlists can be used interchangeably using first and rest.

Streams are lazily computed linked lists



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

    def __init__(self, first, compute_rest=lambda: Stream.empty):
        assert callable(compute_rest), 'compute_rest must be callable.'
        self.first = first
        self._compute_rest = compute_rest

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

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

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

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

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

The way in which names are looked up in Scheme and Python is called *lexical scope* (or *static scope*).

Lexical scope: The parent of a frame is the environment in which a procedure was *defined*. (lambda ...)

Dynamic scope: The parent of a frame is the environment in which a procedure was *called*. (mu ...)

```
> (define f (mu (x) (+ x y)))
> (define g (lambda (x y) (f (+ x x))))
> (g 3 7)
13
```

```
class LetterIter:
    def __init__(self, start='a', end='e'):
        self.next_letter = start
        self.end = end

    def __next__(self):
        if self.next_letter >= self.end:
            raise StopIteration
        result = self.next_letter
        self.next_letter = chr(ord(result)+1)
        return result

class Letters:
    def __init__(self, start='a', end='e'):
        self.start = start
        self.end = end

    def __iter__(self):
        return LetterIter(self.start, self.end)

def letters_generator(next_letter, end):
    while next_letter < end:
        yield next_letter
        next_letter = chr(ord(next_letter)+1)
```

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

```
>>> a_to_c = LetterIter('a', 'c')
>>> next(a_to_c)
'a'
>>> next(a_to_c)
'b'
>>> next(a_to_c)
Traceback (most recent call last):
    ...
StopIteration

>>> b_to_k = Letters('b', 'k')
>>> first_iterator =
b_to_k.__iter__()
>>> next(first_iterator)
'b'
>>> next(first_iterator)
'c'
>>> second_iterator = iter(b_to_k)
>>> second_iterator.__next__()
'b'
>>> first_iterator.__next__()
'd'

>>> for letter in
letters_generator('a', 'e'):
...     print(letter)
a
b
c
d
```

A **table** has columns and rows

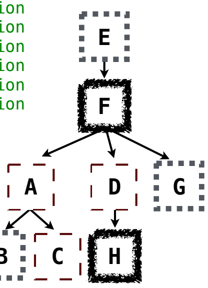
A **column** has a name and a type

Latitude	Longitude	Name
38	122	Berkeley
42	71	Cambridge
45	93	Minneapolis

A **row** has a value for each column

```
select [expression] as [name], [expression] as [name], ... ;
select [columns] from [table] where [condition] order by [order];
```

```
create table parents as
  select "abraham" as parent, "barack" as child union
  select "abraham"          , "clinton"         union
  select "delano"           , "herbert"         union
  select "fillmore"         , "abraham"         union
  select "fillmore"         , "delano"          union
  select "fillmore"         , "grover"          union
  select "eisenhower"       , "fillmore";
create table dogs as
  select "abraham" as name, "long" as fur union
  select "barack"         , "short"       union
  select "clinton"        , "long"        union
  select "delano"         , "long"        union
  select "eisenhower"     , "short"       union
  select "fillmore"       , "curly"       union
  select "grover"         , "short"       union
  select "herbert"        , "curly";
```



```
select a.child as first, b.child as second
  from parents as a, parents as b
  where a.parent = b.parent and a.child < b.child;
```

First	Second
barack	clinton
abraham	delano
abraham	grover
delano	grover

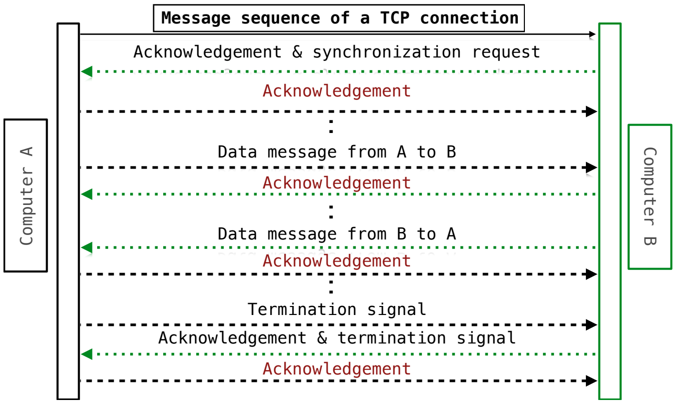
```
with
  ancestors(ancestor, descendent) as (
    select parent, child from parents union
    select ancestor, child
           from ancestors, parents
           where parent = descendent
  )
select ancestor from ancestors where descendent="herbert";
```

ancestor
delano
fillmore
eisenhower

```
create table pythagorean_triples as
  with
    i(n) as (
      select 1 union select n+1 from i where n < 20
    )
  select a.n as a, b.n as b, c.n as c
         from i as a, i as b, i as c
         where a.n < b.n and a.n*a.n + b.n*b.n = c.n*c.n;
```

a	b	c
3	4	5
5	12	13
6	8	10
8	15	17
9	12	15
12	16	20

Message sequence of a TCP connection



CS 61A Final Exam Study Guide – Page 2

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

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

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

```
> (define pi 3.14)              > (define (abs x)
> (* pi 2)                          (if (< x 0)
6.28                                    (- x)
                                        x))
                                > (abs -3)
                                3
```

Lambda expressions evaluate to anonymous procedures.

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

λ

Two equivalent expressions:

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

An operator can be a combination too:

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

In the late 1950s, computer scientists used confusing names.
- **cons**: Two-argument procedure that **creates a pair**
- **car**: Procedure that returns the **first element** of a pair
- **cdr**: Procedure that returns the **second element** of a pair
- **nil**: The empty list

They also used a non-obvious notation for linked lists.
- A (linked) Scheme list is a pair in which the second element is nil or a Scheme list.
- Scheme lists are written as space-separated combinations.
- A dotted list has an arbitrary value for the second element of the last pair. Dotted lists may not be well-formed lists.

```
> (define x (cons 1 2))
> x
(1 . 2)
> (car x)
1
> (cdr x)
2
> (cons 1 (cons 2 (cons 3 (cons 4 nil))))
(1 2 3 4)
```

Not a well-formed list!

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

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

No sign of “a” and “b” in the resulting value

Quotation is used to refer to symbols directly in Lisp.

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

Symbols are now values

Quotation can also be applied to combinations to form lists.

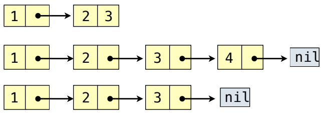
```
> (car '(a b c))
a
> (cdr '(a b c))
(b c)
```

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

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

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

```
> '(1 2 . 3)
(1 2 . 3)
> '(1 2 . (3 4))
(1 2 3 4)
> '(1 2 3 . nil)
(1 2 3)
> (cdr '((1 2) . (3 4 . (5))))
(3 4 5)
```



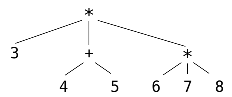
```python
class Pair:
    """A Pair has first and second attributes.

    For a Pair to be a well-formed list,
    second is either a well-formed list or nil.
    """
    def __init__(self, first, second):
        self.first = first
        self.second = second

>>> s = Pair(1, Pair(2, Pair(3, nil)))
>>> print(s)
(1 2 3)
>>> len(s)
3
>>> print(Pair(1, 2))
(1 . 2)
>>> print(Pair(1, Pair(2, 3)))
(1 2 . 3)
```

The Calculator language has primitive expressions and call expressions

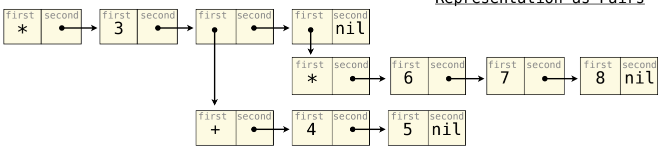
Calculator Expression

```
(* 3
   (+ 4 5)
   (* 6 7 8))
```

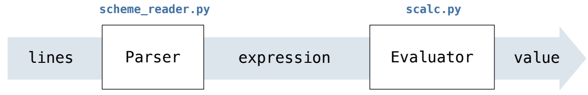
Expression Tree



Representation as Pairs



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



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

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

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

A Scheme list is written as elements in parentheses:

(<element₀> <element₁> ... <elementₙ>)

A Scheme list

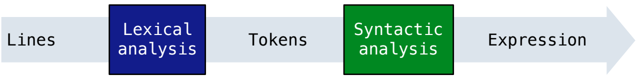
Each <element> can be a combination or atom (primitive).

(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))

The task of *parsing* a language involves coercing a string representation of an expression to the expression itself.

Parsers must validate that expressions are well-formed.

A Parser takes a sequence of lines and returns an expression.



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

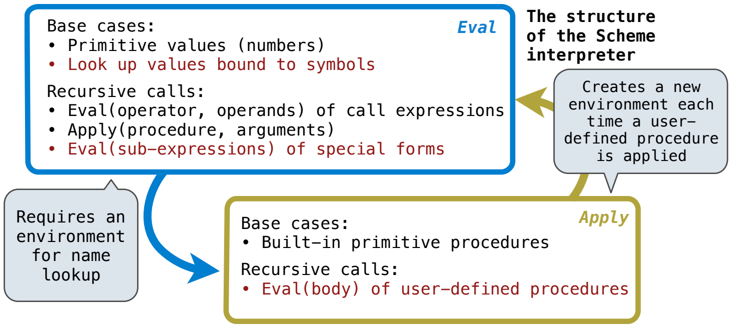
- Iterative process
- Checks for malformed tokens
- Determines types of tokens
- Processes one line at a time

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

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

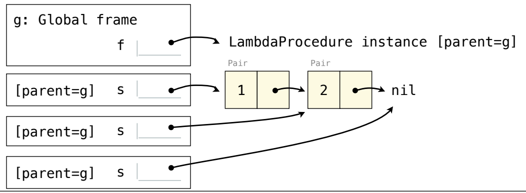
Each call to scheme_read consumes the input tokens for exactly one expression.

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


Eval
Base cases:
• Primitive values (numbers)
• Look up values bound to symbols
Recursive calls:
• Eval(operator, operands) of call expressions
• Apply(procedure, arguments)
• Eval(sub-expressions) of special forms
The structure of the Scheme interpreter
Creates a new environment each time a user-defined procedure is applied
Requires an environment for name lookup
Apply
Base cases:
• Built-in primitive procedures
Recursive calls:
• Eval(body) of user-defined procedures

To apply a user-defined procedure, create a new frame in which formal parameters are bound to argument values, whose parent is the **env** of the procedure, then evaluate the body of the procedure in the environment that starts with this new frame.

```
(define (f s) (if (null? s) '(3) (cons (car s) (f (cdr s)))))
(f (list 1 2))
```



A procedure call that has not yet returned is *active.* Some procedure calls are *tail calls*. A Scheme interpreter should support an unbounded number of active tail calls.
A tail call is a call expression in a *tail context*, which are:
- The last body expression in a **lambda** expression
- Expressions 2 & 3 (consequent & alternative) in a tail context **if** expression

```
(define (factorial n k)                (define (length s)
  (if (= n 0) k                          (if (null? s) 0
    (factorial (- n 1)                     (+ 1 (length (cdr s)))))
               (* k n))))
                                                  Not a tail call
(define (length-tail s)
  (define (length-iter s n)        Recursive call is a tail call
    (if (null? s) n
      (length-iter (cdr s) (+ 1 n)) ) )
  (length-iter s 0) )
```