CS 61A Structure and Interpretation of Computer Programs

Summer 2013 Midterm 1

INSTRUCTIONS

- You have 2 hours to complete the exam.
- The exam is closed book, closed notes, closed computer, closed calculator, except one hand-written 8.5” × 11” crib sheet of your own creation and the official 61A midterm 1 study guide 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
/12	/5	/3	/11	/7	/12	/50

1. (12 points) Proceed with call-tion

For each of the following expressions, write the value to which it evaluates *and* what would be output by the interactive Python interpreter. The first two rows have been provided as examples.

- In the **Evaluates to** column, write the value to which the expression evaluates. If it evaluates to a function value, write FUNCTION. If evaluation causes an error, write ERROR.
- In the column labeled **Interactive Output**, write all output that would be displayed during an interactive session, after entering each call expression. This output may have multiple lines. Whenever the interpreter would report an error, write ERROR. You *should* include any lines displayed before an error.

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

```
from operator import mul

x = 3

def square(x):
    return mul(x, mul(x, 1))

def blaster(y):
    return print(square(y) + x)
```

Expression	Evaluates to	Interactive Output
`square(7)`	49	49
`1/0`	ERROR	ERROR
`square(2) + square(x)`		
`print(square(3))`		
`blaster(5)`		
`print(blaster(2) + 5)`		
`blaster(blaster(3))`		
`25 or (5 / 0)`		

2. (5 points) Lambda? No thanks, I prefer chicken

(a) **(2 pt)** Fill in the blanks below so that `foo(5)(10)()` returns `[5, 10]`. You may *not* write any numbers in your solution, and you may only add expressions in the blanks.

```
foo = lambda __________: lambda y: ______________________________________________
```

(b) **(3 pt)** Fill in the blanks below so that the final call expression below evaluates to a *tuple* value. For this section, you *may* write numbers, but not tuples, and you may only add expressions in the blanks.

```
def love(x):
    if x == 'zedd':
        return [1, 2, lambda: (2, 3)]
    else:
        return lambda: 5

(lambda _____________, banana: foxes_____________________________)(love, 'clarity')
```

3. (3 points) Tracing through the facts

Consider the following portion of code:

```
def tracer(fn):
    def traced(x):
        print('Calling', fn, '(', x, ')')
        result = fn(x)
        print('Got', result, 'from', fn, '(', x, ')')
        return result
    return traced

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

new_fact = tracer(fact)
```

Circle the **Choice X** heading of one of the options below corresponding to what Python would display if we ran `new_fact(2)` in an interpreter session. You may assume that the "ADDRESS" in each output is correct.

Choice A Calling <function fact at ADDRESS> (2) Calling <function fact at ADDRESS> (1) Calling <function fact at ADDRESS> (0) Got 1 from <function fact at ADDRESS> (0) Got 1 from <function fact at ADDRESS> (1) Got 2 from <function fact at ADDRESS> (2) 2	**Choice B** Calling <function fact at ADDRESS> (2) Got 2 from <function fact at ADDRESS> (2) Calling <function fact at ADDRESS> (1) Got 1 from <function fact at ADDRESS> (1) Calling <function fact at ADDRESS> (0) Got 1 from <function fact at ADDRESS> (0) 2
Choice C Calling <function fact at ADDRESS> (2) Got 2 from <function fact at ADDRESS> (2) 2	**Choice D** 2
Choice E The output is none of the above. If you select this choice, please briefly explain why:	

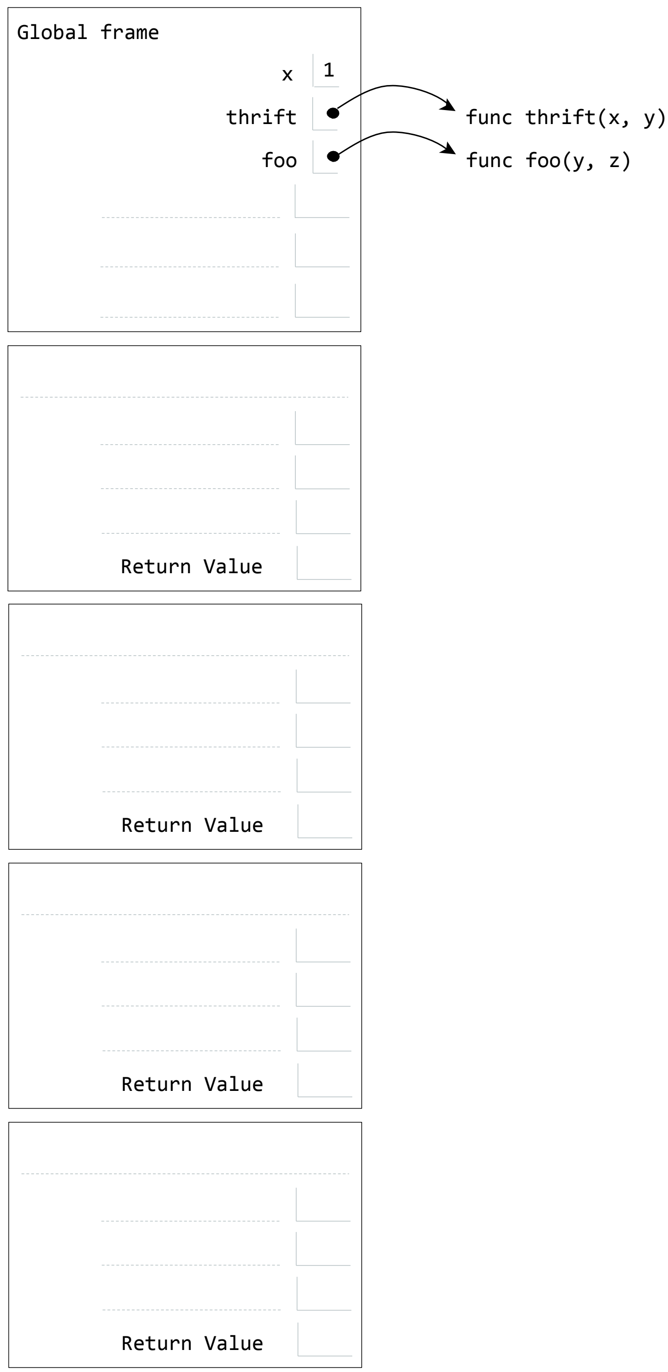
4. (11 points) Save the environment (diagrams)!

(a) (5 pt) Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. You need only show the final state of each frame. *You may not need to use all of the spaces or frames.*

A complete answer will:

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

```
x = 1
def thrift(x, y):
    def inner(z):
        return foo(5, 10) + z
    return inner

def foo(y, z):
    return x + y + z

shop = thrift(2, 3)
shop(7)
```



(b) **(6 pt)** Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. You need only show the final state of each frame. *You may not need to use all of the spaces or frames.*

A complete answer will:

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

```
def make_test(num, checker):
    def test(subm):
        return checker(subm)
    return test

def q5_checker(subm):
    return subm(10) == 15

num = 2
test_q5 = make_test(5, q5_checker)
result = test_q5(lambda x: x + num)
```

5. (7 points) Testing our (pot)luck

While planning the potluck, the 61A staff decided to try and guess the number of people that would show up. In order to do this, they decided to define a new abstract data type to record everyone's predictions. Of course, the 61A staff is bad at computer science, so they need your help to make this work!

(a) (2 pt) We want to make a `prediction` abstract data type that will record both a person's name as well as their guess for the number of attendees. Based on the provided constructor `make_prediction`, fill in the definitions for the `get_name` and `get_guess` selectors.

```
def make_prediction(name, guess):
    return (name, guess)
```

```
def get_name(prediction):
    """Gets the name of the person who made the given prediction.

    >>> get_name(make_prediction('eric', 25))
    'eric'
    """
```

```
def get_guess(prediction):
    """Gets the number of attendees that this prediction expected to show up
    to the potluck.

    >>> get_name(make_prediction('eric', 25))
    25
    """
```

(b) (5 pt) Now complete the `print_winner` function. It takes a sequence of `prediction`s and the actual number of attendees, and prints a congratulatory message based on whose guess was closest. You may assume that the sequence of `prediction`s is non-empty. Ties should go to the person whose `prediction` appears earliest in the sequence. *Remember to respect data abstraction.*

```
def print_winner(predictions, correct_num):
    """Given a sequence of predictions (predictions) and the actual number of
    attendees (correct_num), print the message '___ is the winner', where the
    blank is filled in with the name of the person who made the winning
    prediction.

    >>> albert_pred = make_prediction('albert', 10000)
    >>> brian_pred = make_prediction('brian', 85)
    >>> mark_pred = make_prediction('mark', 97)
    >>> preds = (albert_pred, brian_pred, mark_pred)
    >>> print_winner(preds, 83)
    brian is the winner
    >>> preds2 = (make_prediction('rohan', 90), make_prediction('jeffrey', 70))
    >>> print_winner(preds2, 80)
    rohan is the winner
    """
```

6. (12 points) Learning to count

Steven likes to have a timer with him during lectures so that he knows how much time is left until the end of lecture. He has a timer he really likes that counts the number of seconds that have elapsed since the beginning of lecture.

Unfortunately, it turns out that the timer he bought was manufactured before humans had discovered the number six! The timer works normally, except it skips every number containing a 6 as one of its digits. For example, here are the first twenty numbers displayed by this timer:

`0, 1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 17, 18, 19, 20, 21`

In this example, note how it skips the numbers 6 and 16, because they both have at least one digit that is a 6. This means that when the timer displays 21, in reality only 19 seconds have passed!

Obviously, the way the timer is now isn't very helpful for Steven. Help him solve his problem by writing a function to compute the true number of seconds that have elapsed in his lectures.

For this entire problem, do not use any loop statements. Use recursion only.

(a) **(3 pt)** First, complete the `has_six` helper function, which takes an integer and returns whether or not said integer has a 6 as one of its digits. **Do not use any loop statements. Use recursion. Additionally, do not convert `n` to a string.**

```
def has_six(n):
    """Determines whether the integer n has a 6 as one of its digits.

    >>> has_six(123)
    False
    >>> has_six(567)
    True
    """
```

(b) **(4 pt)** Now, use your `has_six` function to complete the `previous` function. `previous` takes an integer `n` and determines the number that would have appeared *before* it on the timer. In other words, it determines the largest integer less than `n` that does not have a 6 as any of its digits. You may assume that `n` will always be a positive integer. **Once again, do not use any loop statements. Use recursion.**

Note: you may assume that you have a working version of `has_six`. You can receive full credit on this section without completing part (a).

```
def previous(n):
    """Determines the number that showed on the timer just before n.

    >>> previous(3)
    2
    >>> previous(7)
    5
    >>> previous(70)
    59
    """
```

(c) **(5 pt)** Now, use your `previous` function to complete the `num_seconds` function, which takes an integer representing the number shown on the timer and returns the actual number of seconds that have elapsed. **As before, do not use any loop statements. Use recursion.**

Note: you may assume that you have a working version of `previous`. You can receive full credit on this section without completing part (b).

```
def num_seconds(n):
    """Based on the number currently displayed on the timer, n, returns the true
    number of seconds that have elapsed.

    >>> num_seconds(8) # skips 6
    7
    >>> num_seconds(20) # skips 6 and 16
    18
    """
```

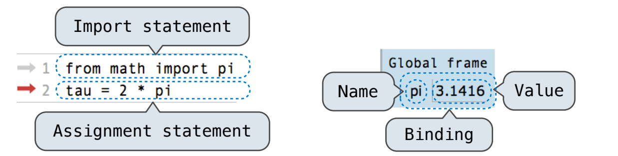
7. (0 points) Extra credit

In the box below, write a positive integer. The student who writes the lowest *unique* integer will receive one extra credit point. In other words, write the smallest positive integer that you think *no one else* will write.

This is the end of the test. Feel free to use the rest of the space for scratch work. You could also draw us a picture, if you're so inclined!

(This page intentionally left blank)

(This page intentionally left blank)



Code:

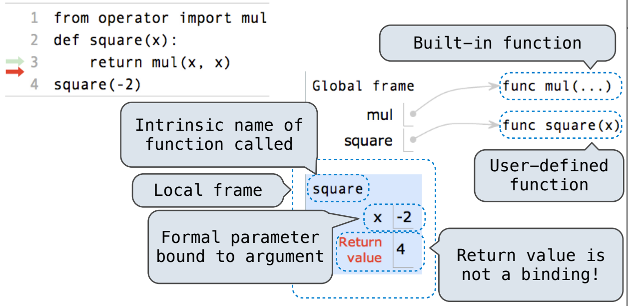
Statements and expressions

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

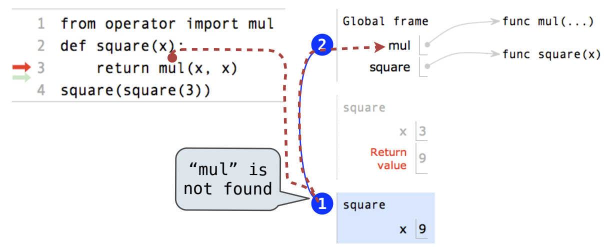
Frames:

A name is bound to a value

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



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:

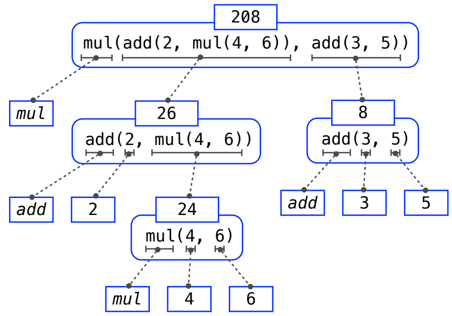
1. Evaluate the subexpression <left>.
2. If the result is a false value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

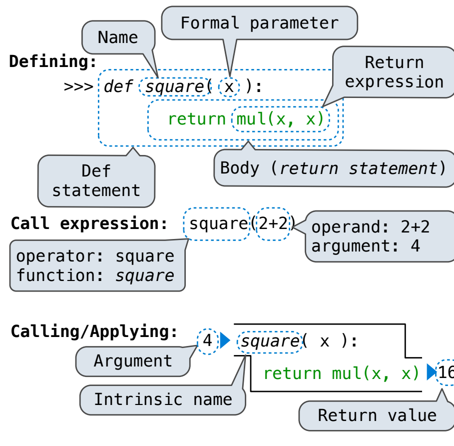
Evaluation rule for not expressions:

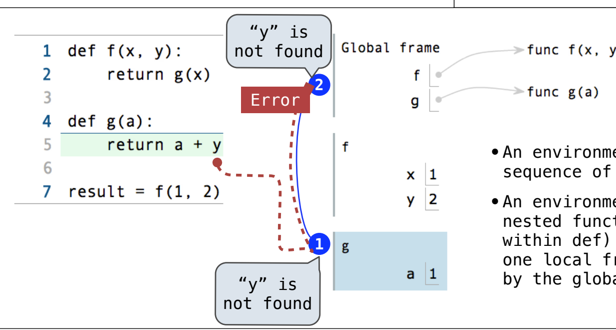
1. Evaluate <exp>; The value is True if the result is a false value, and False otherwise.

Execution rule for while statements:

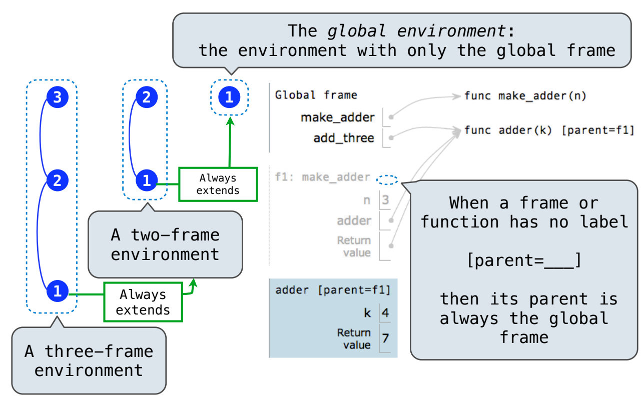
1. Evaluate the header's expression.
2. If it is a true value, execute the (*whole*) suite, then return to step 1.







- 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



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

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

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

    >>> summation(5, cube)
    255
    """
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total
```

Function of a single argument (not called term)

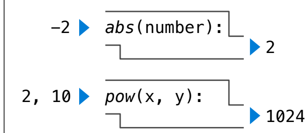
A formal parameter that will be bound to a function

The cube function is passed as an argument value

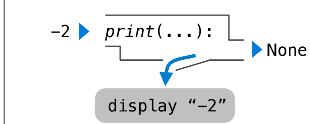
$0 + 1^3 + 2^3 + 3^3 + 4^3 + 5^5$

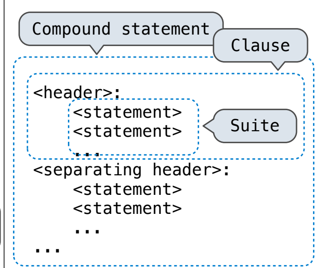
The function bound to term gets called here

Pure Functions



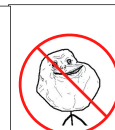
Non-Pure Functions





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

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



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

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

CS 61A Midterm 1 Study Guide — Page 2

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

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

Must be a single expression

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

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

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

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

A function that returns a function

The name add_three is bound to a function

A local def statement

Can refer to names in the enclosing function

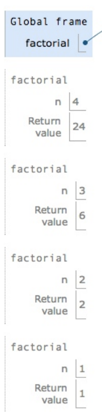
```
make_adder(1)(2)

make_adder(1)   (      2      )
```

Operator — An expression that evaluates to a function value

Operand 0 — An expression that evaluates to any value

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

What does sum_squares need to know about square?
- Square takes one argument. **Yes**
- Square has the intrinsic name square. **No**
- Square computes the square of a number. **Yes**
- Square computes the square by calling mul. **No**



func factorial(n)

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

factorial(4)
```

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

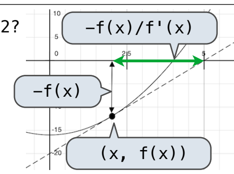
Recursive functions have two important components:
1. *Base case(s)*, where the function directly computes an answer without calling itself
2. *Recursive case(s)*, where the function calls itself as part of the computation

```
square = lambda x: x * x        VS        def square(x):
                                              return x * x
```

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

How to find the square root of 2?

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



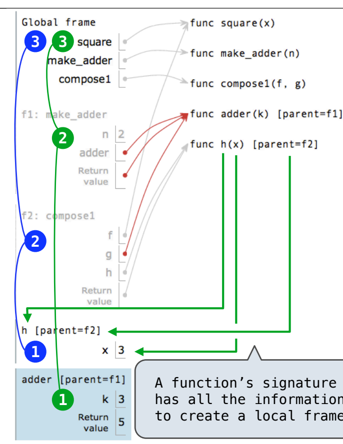
Begin with a function f and an initial guess x

1. Compute the value of f at the guess: f(x)
2. Compute the derivative of f at the guess: f'(x)
3. Update guess to be: $x - \frac{f(x)}{f'(x)}$

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

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

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

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

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

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

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

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

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

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

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



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