CS 61A Structure and Interpretation of Computer Programs

Spring 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" $\times$ 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	Total
/12	/12	/6	/12	/8	/50

1. (12 points) Call Me Maybe

For each of the following call 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 add, mul
def mulled(x, y):
    return mul(x, add(y, x))

def fauxpose(f, g):
    print('maybe')
    def h(x, y):
        f(x, y)
        return g(x, y)
    return h
```

Expression	Evaluates to	Interactive Output
`mulled(5, 5)`	50	50
`1/0`	Error	Error
`mulled(4, 1)`		
`add(mulled(3, 2), print(4))`		
`print(3, print(5, print(1)))`		
`fauxpose(add, mul)(3, 2)`		
`fauxpose(mul, print)(4, 1)`		
`fauxpose(fauxpose, mulled)(2, 5)`		

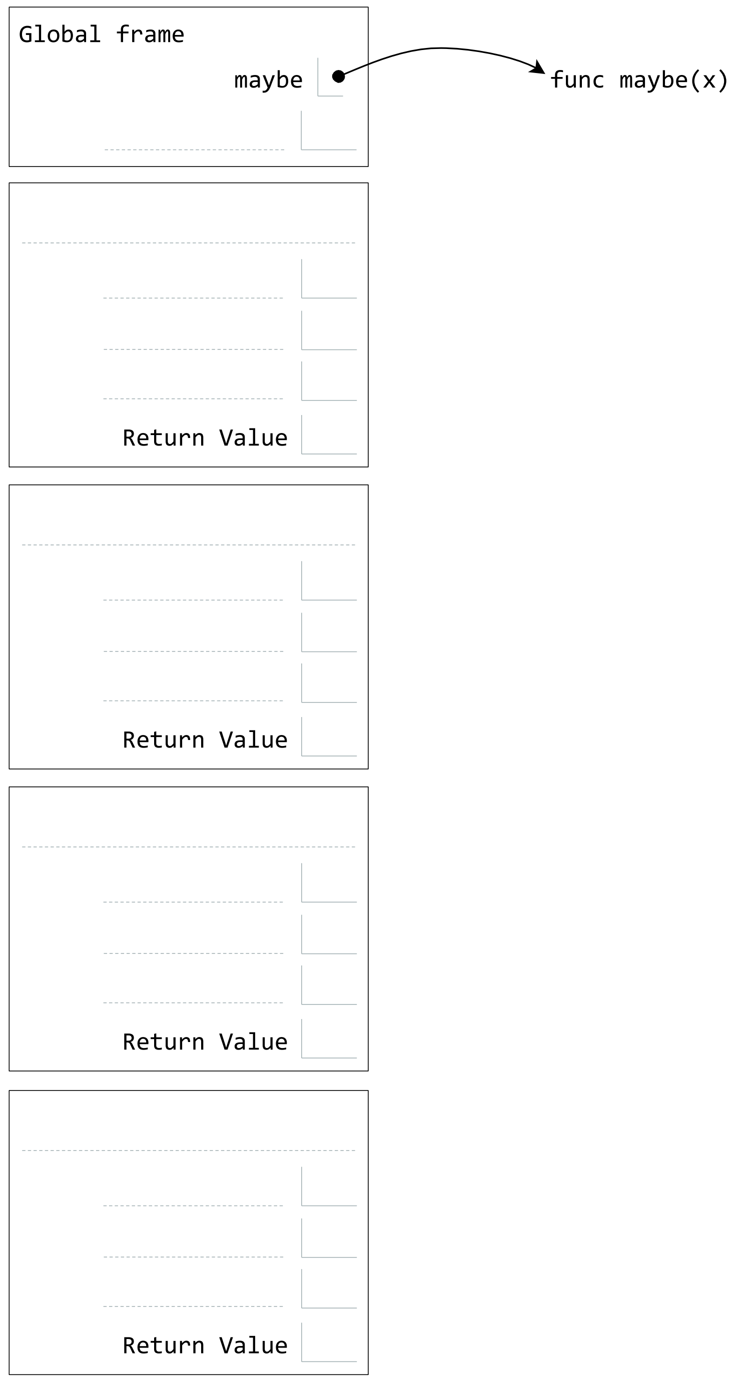
2. (12 points) We Are All Environmentalists

(a) (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 maybe(x):
    next = lambda y: y - 1
    x += 3
    def year(z):
        return next(z + x) * 2
    return year

x = maybe(2)(4)
```

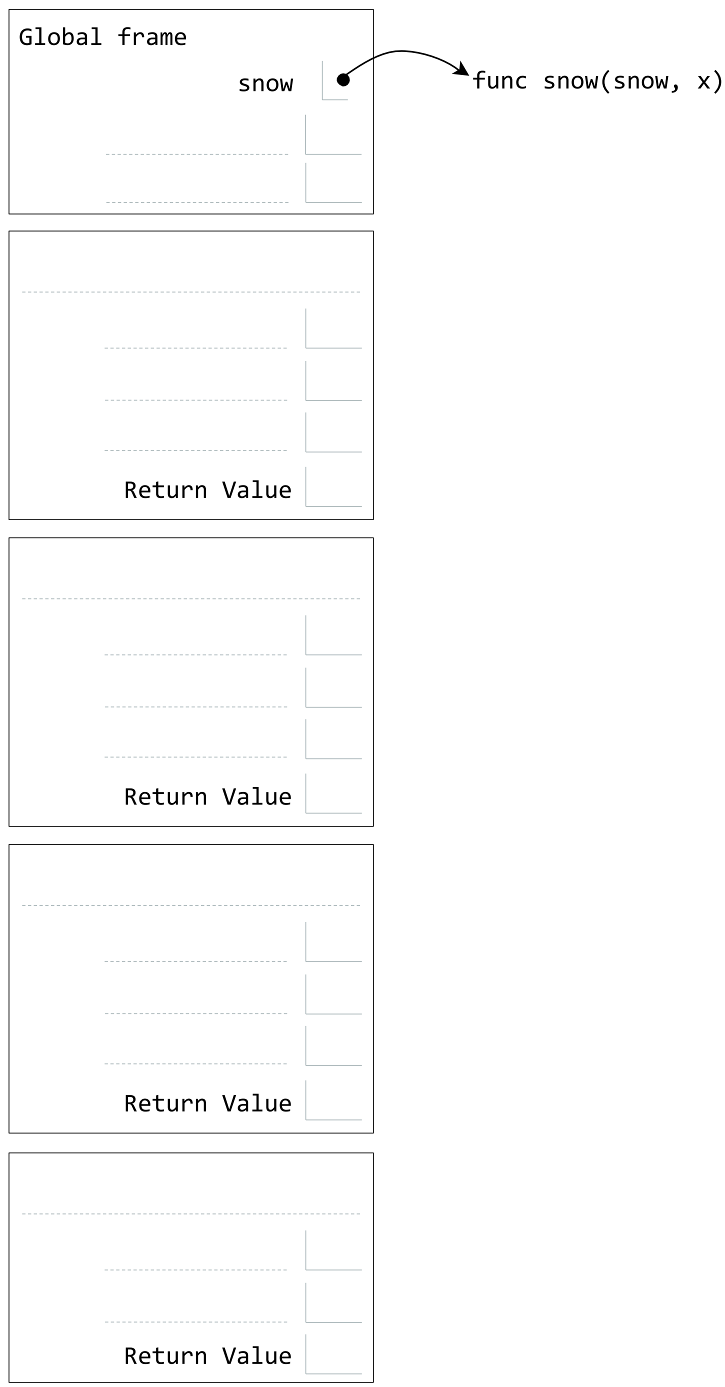


(b) **(6 pt)** Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. You need only show the final state of each frame. *You may not need to use all of the spaces or frames.*

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 snow(snow, x):
    if snow(x, x) == x:
        def x(x):
            return 32
        return x(x)
    else:
        return snow(snow, x)

def flake(x, y):
    return y + x - 1

griffin = snow(flake, 1)
```



3. (6 points) A Higher Order of Protection

Louis Reasoner is making a web application, and he wants to secure it. (Good for him!) One of the ways he wants to secure it is through checking to make sure that the user is an admin when it tries to visit certain confidential pages. So, being a silly programmer, he does the following.

```
def delete_everything(is_admin, request):
    if not is_admin:
        print('ERROR: not admin')
        return
    confirmation = do_bad_stuff(request) # BAD STUFF HAPPENS HERE
    return confirmation

def steal_credit_card_info(is_admin, request):
    if not is_admin:
        print('ERROR: not admin')
        return
    cc_info = hack_a_shaq(request) # DO SOME 1337 HAXORING
    return cc_info
```

However, Alyssa P. Hacker comes across this code, and realizes that there is a better way to do this using higher-order functions! She modifies the above as follows.

```
def delete_everything(request):
    confirmation = do_bad_stuff(request) # BAD STUFF HAPPENS HERE
    return confirmation
delete_everything = protect_me(delete_everything)

def steal_credit_card_info(request):
    cc_info = hack_a_shaq(request) # DO SOME 1337 H4XORING
    return cc_info
steal_credit_card_info = protect_me(steal_credit_card_info)
```

Help her to complete the code by filling in the function below. The new code should provide the same functionality as the original code. For example, calling `delete_everything(True, my_request)` should have the same effect in both versions of the code.

You may leave lines blank if you do not need them.

```
def protect_me(fn):

    -------------------------------------------------------------------

        if not is_admin:
            print('ERROR: not admin')
            return

        return _______________________________________________________

    -------------------------------------------------------------------
```

4. (12 points) Da Visors Provide Protection from Puns

An integer d is a *divisor* of another integer n if it evenly divides n, i.e. the remainder is 0 when dividing n by d. The divisors of n include 1 and n itself.

(a) Complete the function below to compute the number of positive divisors of a positive integer. Fill in the blanks. You may leave a line blank if you do not need it.

```
--------------------------------------------------------------------------

def num_divisors(n):
    """Computes the number of positive divisors of a positive integer.

    >>> num_divisors(4)    # 1, 2, and 4
    3
    """
    i, count = 1, 0

    while ____________________________________________________________:

        if ____________________________________________________________:

            count = count + 1

        ____________________________________________________________

    return count
```

(b) Write a function that computes the sum of the positive divisors of a positive integer.

```
def sum_divisors(n):
    """Computes the sum of the positive divisors of a positive integer.

    >>> sum_divisors(4)    # 1 + 2 + 4
    7
    """
```

(Continued on next page)

(c) A positive integer n is called *abundant* if the sum of its divisors (except n itself) is strictly larger than n. It is called *perfect* if the sum of its divisors (except n itself) is exactly equal to n. Finally, n is deficient if the sum of its divisors (excluding n) is strictly less than n. Write a function that returns the string `'abundant'` if the input `n` is abundant, `'perfect'` if `n` is perfect, and `'deficient'` if `n` is deficient. You may call `sum_divisors` and assume that it works correctly.

```
def describe(n):
    """Returns whether n is abundant, perfect, or deficient.

    >>> describe(4)    # 1 + 2 < 4
    'deficient'
    """
```

5. (8 points) Lambda the Free

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

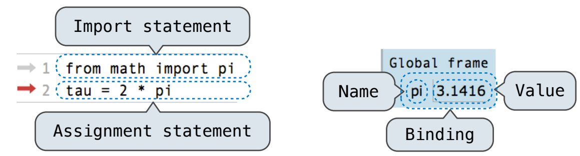
```
square = lambda x: x * x

def muckluck(x):
    square(square(x))

def apply(f, x):
    return f(x)

def apply_many(f, x, n):
    while n > 0:
        x = f(x)
    return x

def wut(f):
    return lambda f: f(x)

def pair(x, y):
    return lambda k: x if k == 0 else y
```

For each of the following call expressions, write the value to which it evaluates. If the value is a function value, write FUNCTION. If evaluation causes an error, write ERROR. If evaluation would run forever, write FOREVER.

Expression	Evaluates to
`square`	
`muckluck(2)`	
`apply(pair, 3)`	
`wut(square)`	
`pair(3, 4)(1)`	
`pair(apply, square)(0)`	
`apply_many(square, 3, 0)`	
`apply_many(square, 3, 2)`	

(This page intentionally left blank)

(This page intentionally left blank)



Code:

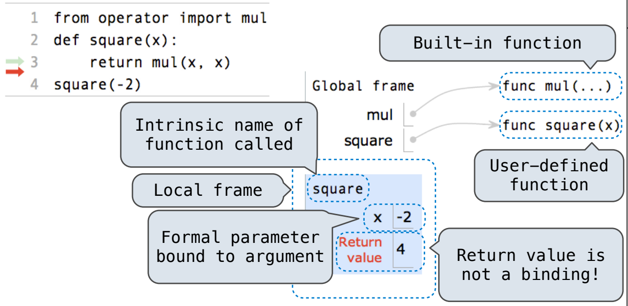
Statements and expressions

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

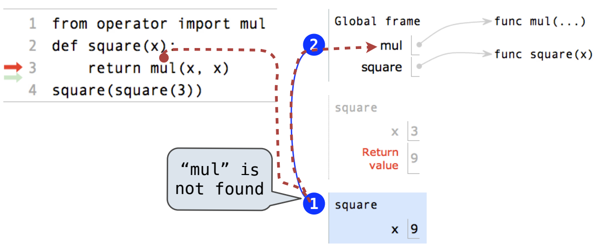
Frames:

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:

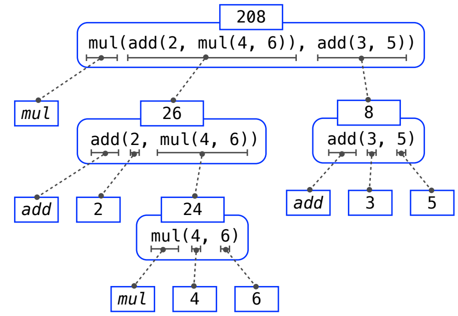
1. Evaluate the subexpression <left>.
2. If the result is a false value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

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.



Pure Functions

-2 ▶ *abs*(number): ▶ 2

2, 10 ▶ *pow*(x, y): ▶ 1024

Non-Pure Functions

-2 ▶ *print*(...): ▶ None

display "-2"

Defining:

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

Name · Formal parameter · Return expression · Def statement · Body (*return statement*)

Call expression: square(2+2)

operator: square
function: *square*

operand: 2+2
argument: 4

Calling/Applying:

4 ▶ *square*(x): return mul(x, x) ▶ 16

Argument · Intrinsic name · Return value

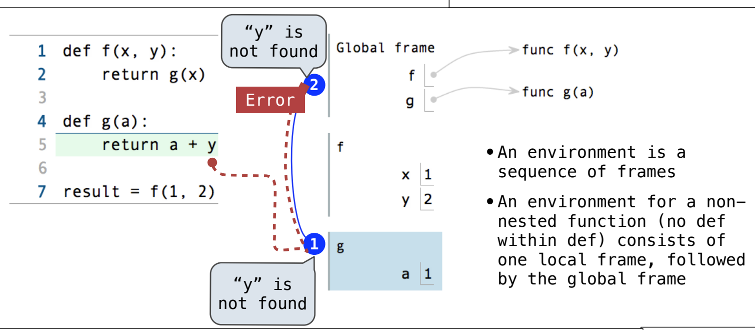
Compound statement · Clause

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

Suite

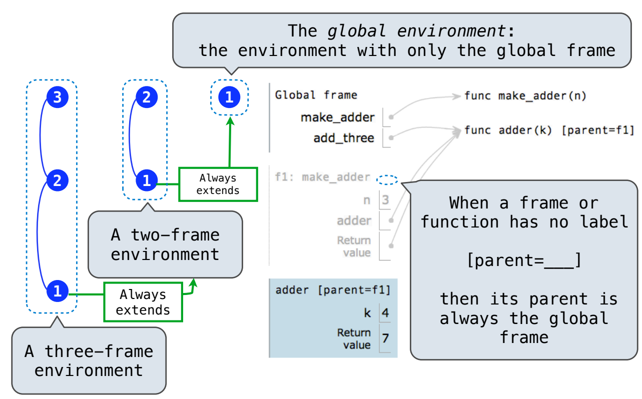
1 statement,
3 clauses,
3 headers,
3 suites,
2 boolean contexts

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

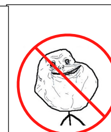


• An environment is a sequence of frames

• 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



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

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

Function of a single argument (not called term)

```
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
```

A formal parameter that will be bound to a function

The cube function is passed as an argument value

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

The function bound to term gets called here

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)

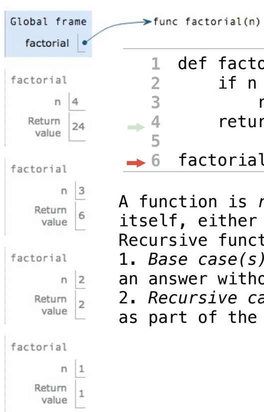
Operator — An expression that evaluates to a function value

Operand 0 — An expression that evaluates to any value

```
def square(x):
    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**




```
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
```

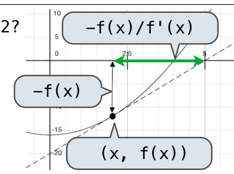
VS

```
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
```

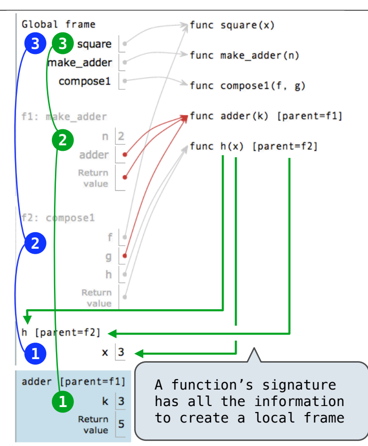


Begin with a function f and an initial guess x

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