Welcome to CS61A Disc 2 Sect. 29/47 :D

Dickson Tsai
dickson.tsai+cs61a@berkeley.edu
OH: Tu, Th 4-5pm 411 Soda

Previous: Expressions, Statements and Functions **>>>**

Today: Higher order functions >>>

Next stop: Recursion

Section Quiz Logistics

- Please take out a piece of paper
- Put your name, login (e.g. cs61a-ty), SID, and section number (#47 for 11-12:30 pm, #29 for 6:30-8pm)
- Graded on effort: effort = revise your quiz with diff color
- Before we begin, please answer:
  - What can I do to make your discussion experience better?
  - Which topics do you feel most/least confident in?
- If you are bored, you can answer:
  - Anything interesting: poem, art, code, fun fact, anecdote

Env. Diagram Review

- Draw an environment diagram for the following code:

```
def square(x):
    return x * x
square(2)
```

Rules we need:

- def statement rules
- Eval call expression
  - Eval operator, operands
  - Apply func to args
    - Create new frame
    - Bind formals to args
    - Evaluate body w/ new frame as current frame
- Name lookup

Tips:

- You should label every parent, even if the parent is the global frame
- It may be helpful to keep track of the current environment (a "stack" of frames)
  - Add to the stack when you create a new frame
  - Pop from the stack when you return from a frame

Section Quiz

- Draw an environment diagram for the following code:

```
a = 2
def foo(x=2):
  a = 1
  return a + x
def bar(x):
  return a + x
print(foo(), bar(1))
```

Tips:

- You should label every parent, even if the parent is the global frame
- It may be helpful to keep track of the current environment (a "stack" of frames)
  - Add to the stack when you create a new frame
  - Pop from the stack when you return from a frame

Section Quiz - Advanced

- Draw an environment diagram for the following code:

```
b = 2
def a(a):
  def b(a):
    print(b)
    return a
  b, a = a, b
  return a(a)
print(a(1)(3))
```

Tips:

- You should label every parent, even if the parent is the global frame
- It may be helpful to keep track of the current environment (a "stack" of frames)
  - Add to the stack when you create a new frame
  - Pop from the stack when you return from a frame

Optional Section Homework [Canceled]

- This will be used to mark attendance for the next section for midterm recovery. Will help participation (but you can participate in other ways as well!)
- These should take at most 15 minutes, if you keep up with your reading.
- Submit with voice recording, email, paper, slideshow, however you want.
- Graded on effort, but use your time wisely. 10 minutes of BS < 10 minutes of ***citing*** the textbook
- Feel free to make your own questions!

- New policy
  - In-class quizzes graded on effort to mark discussion attendance
  - Please still be prepared for discussion.
  - Those who submitted this assignment by Thursday 9/4 will have a free absence from discussion for midterm recovery.

Anuncios

- Project 1 due 11:59 PM next Wednesday 9/17
  - START NOW!!!
- Take-home quiz posted 3pm this Wednesday, due 11:59PM tonight 9/11
  - To see if you are on pace with the course
- Reminder: My OH are Tuesdays, Thursdays 4-5 PM in Soda 411
- Guerrilla Section: A fancy extra section where you work with others passionate people on more practice problems. Saturday 12:30-3pm in Soda 306.

Tip of the Day

- Come up with simple examples!
  - What's the simplest example of a function?
    - ```
      def square(x):
          return x * x
      ```
  - What's the simplest example of a HOF?

```
def foo(x):
  def bar():
    return x
  return bar
```
```

Tip of the Day

- Come up with simple examples!
  - But Dickson, this seems a bit stupid, right?
  - Well, if you are bored, you must be ready to take on new challenges. Combine with other concepts you've learned: e.g. add conditionals, prints, etc.
  - Hey, now you have a working banking system!

```
def make_withdraw(balance):
  def withdraw(amount):
   if amount > balance:
     print("Insufficient funds")
   else:
     balance -= amount
     print("Here is your ${0}".format(amount))
 return withdraw
```

What are boolean operators???

- Boolean data type vs. numbers:
  - {True, False} (two possible values) vs. (-∞, ∞)
- Primarily built-in
- Either:
  - Create boolean contexts (convert all values to booleans), e.g. and, or, not
    - False values: 0, '', [], {}, False,
    - True values: Anything else
  - Result is a boolean
    - E.g. ==, !=, >, <, is, in, …

Pop Quiz – Boolean operators

- Is -1 a False value in a boolean context?
  - No
- Can you make expressions with boolean operators? Give an example.
  - Absolutely! 6 > 5 is a expression. Remember that expressions compute to a value.

  - With good names, you can make intuitive expressions!
    - return dickson not in alumni_list and is_student(dickson) and dickson.age >= 21

Explore Boolean Operators

subexp 1 **or** _subexp2_

Exp1	Exp2	RV
T	T	T
T	F	T
F	T	T
F	F	F

subexp 1 **and** _subexp2_

Exp1	Exp2	RV
T	T	T
T	F	F
F	T	F
F	F	F

not _subexp1_

Exp1	RV
T	F
F	T

Example

subexp 1 **or** _subexp2_

Exp1	Exp2	RV
T	T	T
T	F	T
F	T	T
F	F	F

5 **or** 1 / 0

or is the operator

5 is the operand -> **True** in Boolean context

True or 1 / 0

Since True in Exp1 guarantees True RV, the value of the expression `5 or 1 / 0` is **True**

[Discuss] Walk through step by step

subexp 1 **and** _subexp2_

Exp1	Exp2	RV
T	T	T
T	F	F
F	T	F
F	F	F

a, b = 10, 6
a != 0 **and** b > 5

Boolean Takeaways

1. Now that you know about boolean operators, just evaluate the operands!
2. What's the difference from the regular call expression procedure?
    1. You sometimes do not evaluate all of the operands!!!

Break

Iteration Important Q's

1. What do you want to repeat in your loop?
2. What vars do you want to keep track of?
3. Where do you want to start/stop for each var?

Thinking about these questions *before you code* will save you a lot of time.

You can see that once you answer these questions thoroughly, the code follows directly (watch the colors in the next few examples).

Iteration Important Qs - Example

1 + 2 + 3 + … + n

Break this up into n pieces of computation:

1 # + 2 # + 3 # + 4 # … # + n

1. What do you want to repeat in your loop?

total = total + current element (do something w/ current info)

current element = current element + 1 (get to next piece of info)

2. What vars do you want to keep track of?
3. Where do you want to start/stop for each var?

	Start	Stop
total	0	Return value
current	1	n

Iteration Important Qs - Example

1. What do you want to repeat?

   total = total + current element

   current element = current element + 1

2. What vars do you want to keep track of?
3. Where do you want to start/ stop for each var?

	Start	Stop
total	0	Return value
current	1	n

```
def sum(n):
  total, current = 0, 1
  while current <= n:
    total += current
    current += 1
  return total
```

Iteration Important Qs - Example

Factorial: What changes? 1 * 2 * … * n

Break this up into n pieces of computation:

1 # * 2 # * 3 # * 4 # … # * n

1. What do you want to repeat in your loop?

total = total * current element (do something w/ current info)

current element = current element + 1 (get to next piece of info)

2. What vars do you want to keep track of?
3. Where do you want to start/stop for each var?

	Start	Stop
total	1	Return value
current	1	n

Iteration Important Qs - Example

1. What do you want to repeat?

   total = total * current element

   current element = current element + 1

2. What vars do you want to keep track of?
3. Where do you want to start/ stop for each var?

	Start	Stop
total	1	Return value
current	1	n

```
def fact(n):
    total, current = 1, 1
    while current <= n:
        total *= current
        current += 1
    return total
```

Iteration Important Qs - Example

Take in user input, determine smallest number.

Break this up into n pieces of computation:

Input, set min # Input, set min if smaller # input, set min if smaller…

1. What do you want to repeat in your loop?

Check if current num is smaller than our stored minimum

Get the next value

2. What vars do you want to keep track of?
3. Where do you want to start/stop for each var?

	Start	Stop
buffer	First input	Empty
current	None	Return value

Iteration Important Qs - Example

1. What do you want to repeat?

Check if current num is smaller than our stored minimum

Get the next value from user

2. What vars do you want to keep track of?
3. Where do you want to start/ stop for each var?

	Start	Stop
buffer	First input	Empty
current	None	Return value

```
def find_min():
    buf = input(“>”)
    curr_min = None
    while buf is not None:
        new_num = int(buf)
        if curr_min is None \
                or new_num < curr_min:
            curr_min = new_num
        buf = input(“>”)
    return total
```

Iteration Takeaways

1. The header gives you a lot of control!! You can manipulate the computer's flow based on judicious choice of header
2. For while loops: may be helpful to keep track of values so you don't lose track of where you are/run into an infinite loop
    1. Q: How do you keep track of values???
    2. A: Use names!! E.g. counter = 1. Now, you can update your counter in the loop like counter += 1
3. Think about what you are repeating.
4. Outline:
    1. What do you want to repeat in your loop?
    2. What vars do you want to keep track of?
    3. Where do you want to start/stop for each var?

Break

Characteristics of Functions

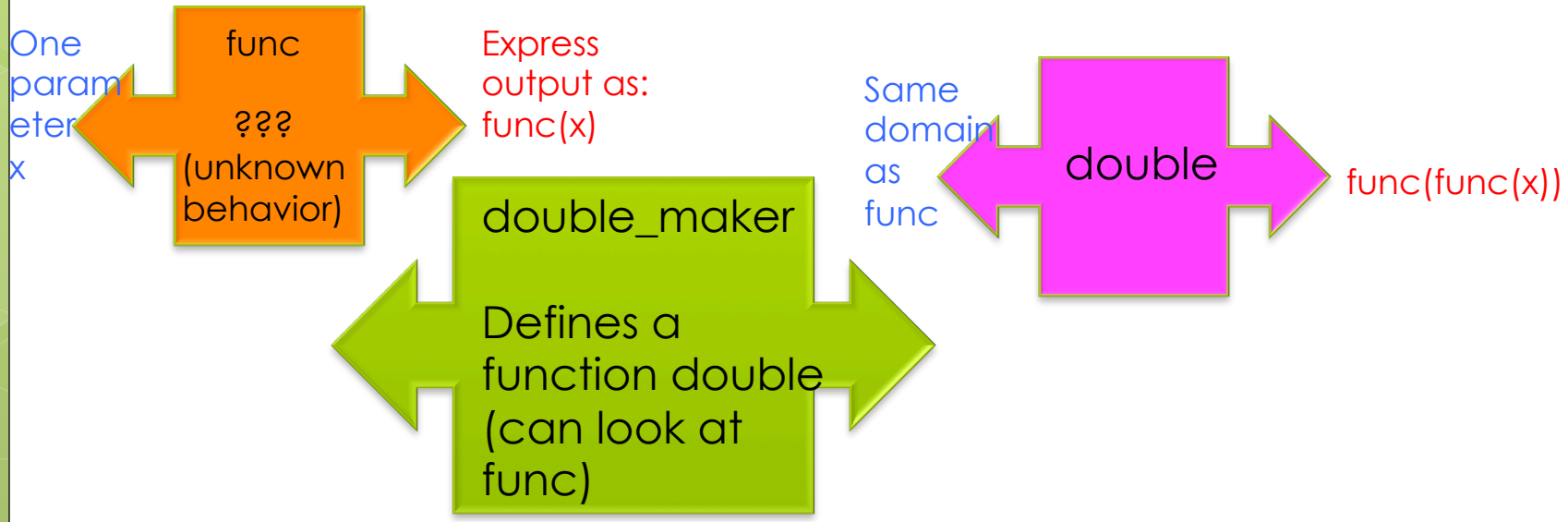
- Recall that a function has 3 characteristics: domain, range, and behavior
- Recall from Lab 2 (this is similar but not the same as make_buzzer):
- "Write a function `double_maker` that takes in a one-argument function `func` and returns a function (double) that takes in one argument and returns the result of calling `func` twice on that argument"
- [Discuss] What is the domain, range, and behavior of double_maker? What is a best way to draw all that information out visually?

Characteristics of Functions

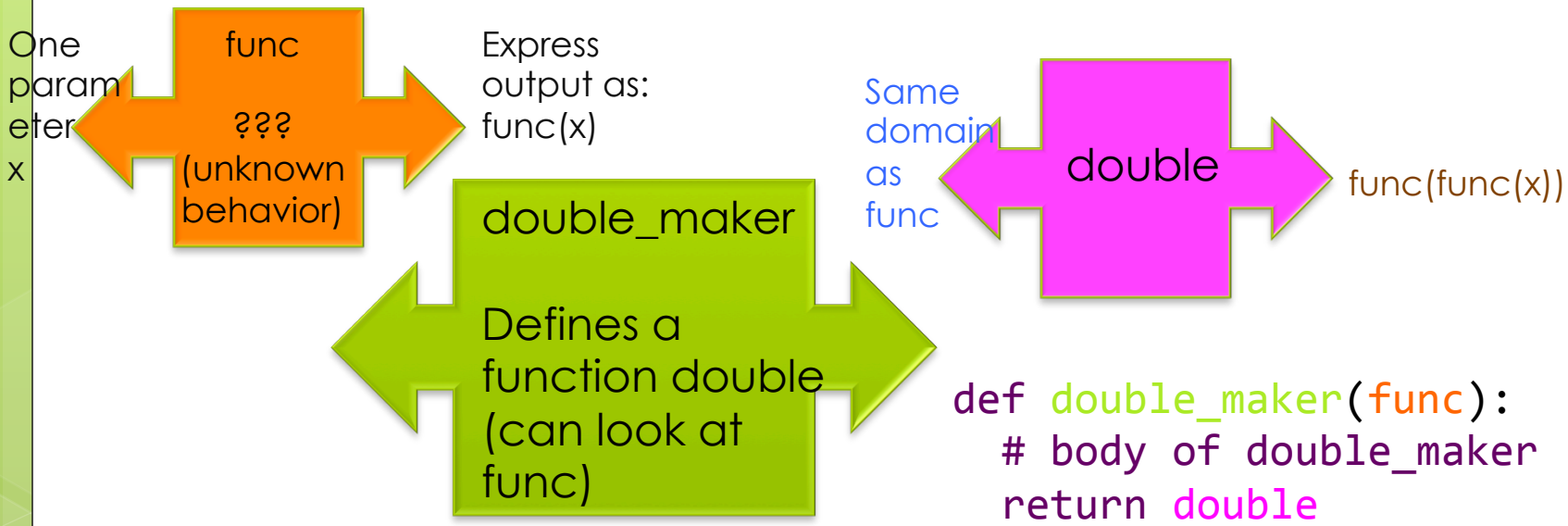
- "Write a function `double_maker` that takes in a one-argument function `func` and returns a function (double) that takes in one argument and returns the result of calling `func` twice on that argument"
- Domain: a one-argument function func
- Behavior: define a function that …
- Range: that defined function
  - Domain of returned fn: same domain as domain of func
  - Behavior: Call func twice on the input
  - Range: The value of that double function call

Characteristics of Functions

- "Write a function `double_maker` that takes in a one-argument function `func` and returns a function (double) that takes in one argument and returns the result of calling `func` twice on that argument"
- Best way to visualize:

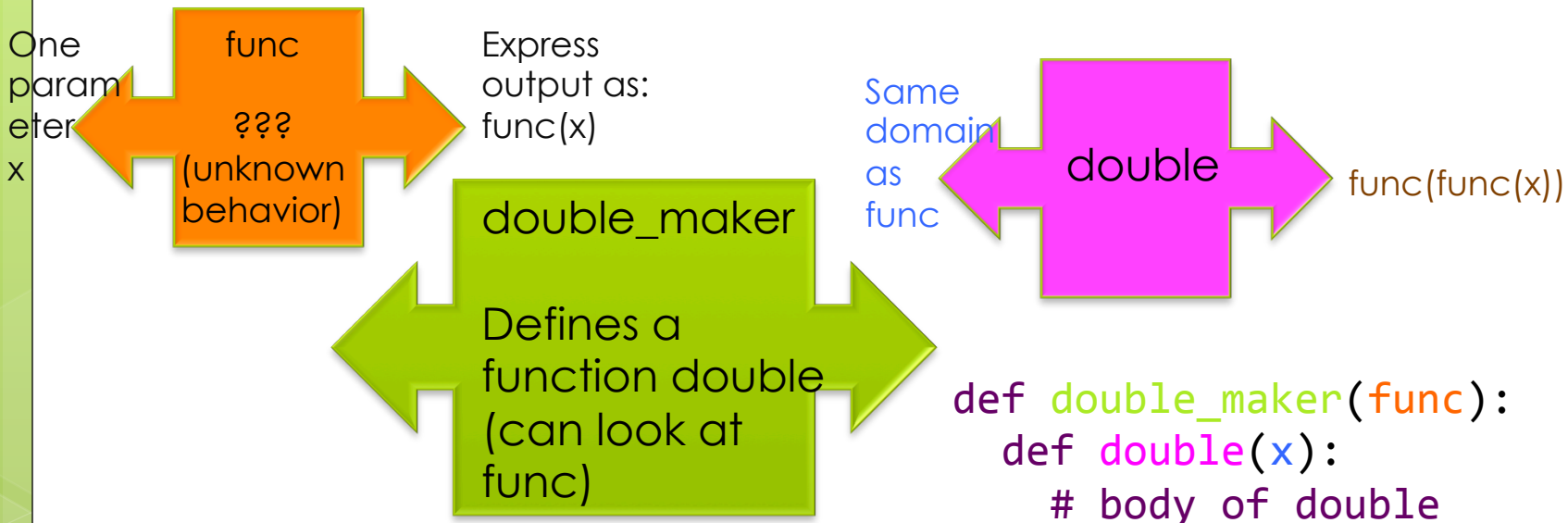


Characteristics of Functions




```
def double_maker(func):
    # body of double_maker
    return double
```

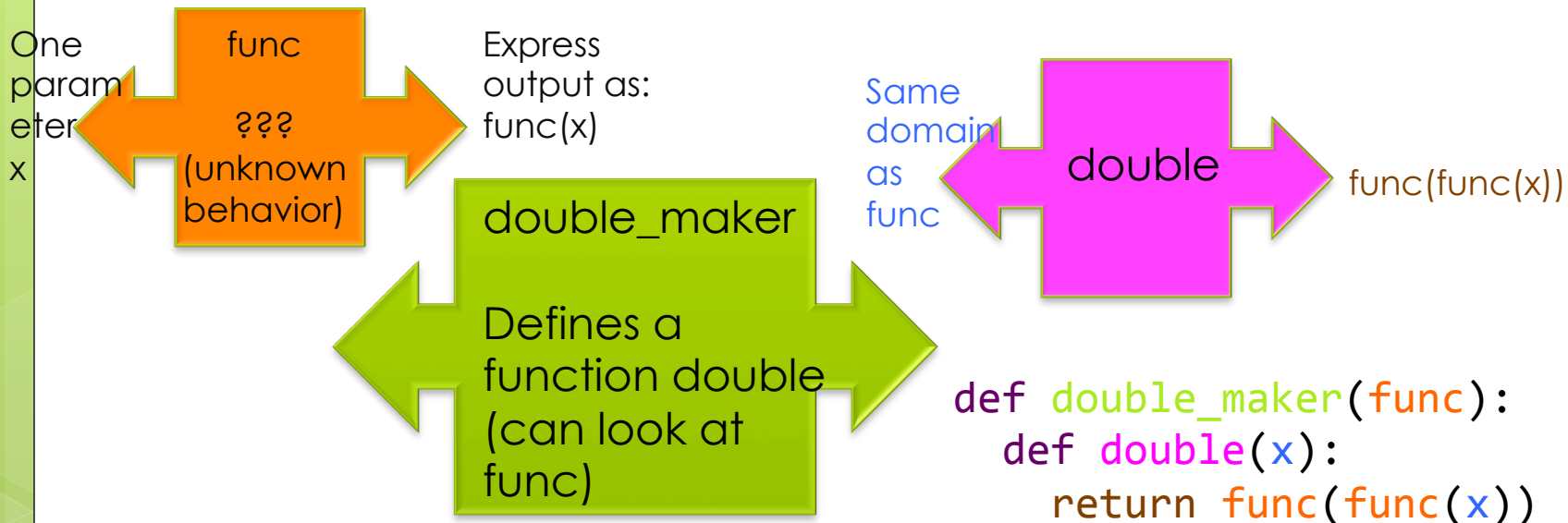
Characteristics of Functions




```
def double_maker(func):
  def double(x):
    # body of double
  return double
```

Even though getting the domain/range right might not seem to be important, it's an important step in the right direction.
It's like aiming your basketball shot. An errant shot has no chance of going in. A straight shot always has more of a chance.

Characteristics of Functions




```
def double_maker(func):
    def double(x):
        return func(func(x))
    return double
```

double_maker can now make functions that we can customize. It combines the power of argument passing and function definitions!

Now: try writing call expressions for double_maker and its output and draw environment diagrams!

HOF Takeaways

1. If you define a function within another function:
   1. You can take in arguments that your inner function can access
   2. Consequence: your inner functions are customizable!
2. If you take in a function as an argument:
   1. You can apply any operation to the data you have!
3. These advantages should not be memorized, but rather appreciated as you go along. Get an intuitive feel for these HOFs.
4. They are not the solution to everything. We will see other solutions later, like OOP.

Break

Env Diagram Rules

Creating a function using def:

1. Draw func <name>(<param1>, <param2>, …) [parent=?]
2. The parent of the function is the current frame
3. For def only: bind this function to the same name in the current frame

Calling user-defined functions:

1. Evaluate the operator and operands
2. Create a new frame. Label the frame with the info from the function object <name> [parent=?]
3. Bind the formal parameters to the arguments (values of operands)
4. Evaluate the body with new frame as current frame
5. Return back to frame that called the function

Assignment

1. Evaluate the RHS 2. Bind name on LHS to value on RHS in current frame

Lookup

1. Start at current frame. If name is there, great! If not, go to the **parent** frame

Env Diagram Rules

1. Don't follow your intuition, because you have never seen rules like these (unless you programmed before, kudos!)
2. Follow these rules until you get the right intuition.
3. Env diagrams are fun! They just take time to master.
4. Some sacrifice now will be rewarding in the long run ☺
5. Now let's look at the rules again.

Env Diagram Rules

Creating a function using def:

1. Draw func <name>(<param1>, <param2>, …) [parent=?]
2. The parent of the function is the current frame
3. For def only: bind this function to the same name in the current frame

Calling user-defined functions:

1. Evaluate the operator and operands
2. Create a new frame. Label the frame with the info from the function object <name> [parent=?]
3. Bind the formal parameters to the arguments (values of operands)
4. Evaluate the body with new frame as current frame
5. Return back to frame that called the function

Assignment

1. Evaluate the RHS 2. Bind name on LHS to value on RHS in current frame

Lookup

1. Start at current frame. If name is there, great! If not, go to the **parent** frame

Optional Conceptual Questions (Wednesday 9/10 lecture)

1. What's the purpose of drawing arrows to function objects everywhere on the environment diagram? Hint: Consider what happens when you want to rebind names.
2. How do we determine the parent of a function object?
3. Can there be more than one frame referring to the same function object? (e.g. have twice: square [parent=f1])
4. How do you know if an expression is a call expression? (Should be a very simple question)
5. How is the return statement of a function related to a call expression involving that function?
6. Why do we bother with frames? What's the problem with having all the variables and action in the global frame?