Public-Service Announcement

“Berkeley Consulting is a student-run consulting group on campus. We are a group of 30 students that complete 4 projects a semester for Fortune 500 firms, startups, and nonprofit organizations. We solve problems for and provide solutions to companies from all industries like Google, Dropbox, and Khan Academy. We are currently recruiting and would love to have you join us! We are looking for students from all majors who are driven, critical thinkers, team players, and able to think outside the box. If you are interested in joining us, please visit bc.berkeley.edu for more information and come to one of our workshops on January 24th and 26th to learn more about our group. We hope to see you at one of our events next week!”

From Last Time

• From last lecture:
  - Values are data we want to manipulate and in particular,
  - Functions are values that perform computations on values.
  - Expressions denote computations that produce values.
  - Today, we'll look at them in some detail at how functions operate on data values and how expressions denote these operations.
  - As usual, although our concrete examples all involve Python, the actual concepts apply almost universally to programming languages.

Functions

• For this lecture, we're going to use this notation to show function values (which are created by evaluating function definitions):
  - abs(number)
  - add(left, right)

• The green parenthesized lists indicate the number of parameter values or inputs the functions operate on (this information is also known as a function's signature).
• For our purposes, the blue name is simply a helpful comment to suggest what the function does, and the specific (green) parameter names are likewise just helpful hints.
• (Python actually maintains this intrinsic name and the parameter names internally, but this is not a universal feature of programming languages.)

Pure Functions

• The fundamental operation on function values is to call or invoke them, which means giving them one value for each formal parameter and having them produce the result of their computation on these values:
  - abs(-5)
  - add(29, 13)

• These two functions are pure: their output depends only on their input parameters' values, and they do nothing in response to a call.

Impure Functions

• Functions may do additional things when called besides returning a value.
• We call such things side effects.
• Example: the built-in print function:
  - print(-5)

• Displaying text is print's side effect. Its value, in fact, is generally the null value.
• From last lecture: Values are data we want to manipulate and in
Other Kinds of Impurity

- Most side-effects involve changing the value of some variable.
- Example: the function `random.randint`:

```python
>>> random.randint(0, 100)
# Random number in 0--100.
13
>>> random.randint(0, 100)
55
# Something must have changed!
```

Call Expressions

- A call expression denotes the operation of calling a function.
- Consider `add(2, 3)`:
  - The operator and the operands are all themselves expressions (recursion again).
  - To evaluate this call expression:
    - Evaluate the operator (let's call the value `C`);
    - Evaluate the operands in the order they appear (let's call the values `P_0` and `P_1`);
    - Call `C` (which must be a function) with parameters `P_0` and `P_1`.

Together with the definitions for base cases (arity literal expressions and symbolic names), this describes how to evaluate any call.

Example: From Expression to Value

Let's evaluate the expression `mul(add(2, mul(0x4, 0x6)), add(0x3, 0x5))`.
In the following sequence, values are shown in boxes.
Everything outside a box is an expression.

1. `mul(add(2, mul(0x4, 0x6)), add(0x3, 0x5))`
2. `mul(left, right)`
3. `mul(left, right)`
4. `mul(left, right)`
5. `mul(left, right)`
6. `mul(left, right)`

Example: Print

- What about an expression with side effects?
- `print(print(1), print(2))`
- `(None, print(print(2)))`
- `(None, (None, print('2')))`
- `(None, None)` and print 'None None'.

Names

- Evaluating expressions that are literals is easy, the literal's text gives all the information needed.
- How did I evaluate names like `add`, `mul`, or `print`?

Deduction: there must be another source of information.

We'll first try a simple approach: substitution of values for names.
This won't cover all the cases, however, and so we'll introduce the concept of an environment.

Example: The function `random.randint`

```python
>>> random.randint(0, 100)
# Random number in 0--100.
```

# Random number in 0--100.

```python
>>> random.randint(0, 100)
55
# Something must have changed!
```
Substitution

Let's try to explain the effect of

\[ x = 3 \]

\[ y = x \times 2 \]

\[ z = y ^ {x} \]

by treating each assignment (=) as a definition.

Thus, we get

\[ x = 3 \]

\[ y = 3 \times 2 \]

\[ y = 6 \]

\[ z = y ^ {3} \]

\[ z = 6 ^ {3} \]

\[ z = 216 \]

That is, we replace names by their definitions (values).

Substitution and Functions

Now consider a simple function definition:

```python
def compute(x, y):
    return (x * y) ** x
```

```python
print(compute(3, 2))
```

A `def` statement is sort of like an assignment, but specialized to functional values.

The `def` statement above defines `compute` to be "the function of \( x \) and \( y \) that returns \( (x \times y)^x \)."

Here, I'll use a common notation for that (due to Church):

\[ \lambda x, y : (x \times y)^x \]

So after substitution for `compute`, we have

```python
print((\lambda x, y : (x \times y)^x)(3, 2))
```

What about this?

```python
def incr(n):
    def f(x):
        return n + x
    return f

print(incr(5)(6))
```

The `incr` function returns a function. The argument to `print` then calls this function on 6.

Now what?

Here, we use a common notation for that (due to Church):

\[ \lambda x : \lambda y : ((x + y)^x) \]

And if for \( x \)

\[ x = 5\]

\[ y = 2 \]

Then \( x = y = 5 \)

If we just substitute for the first \( x \) as before:

\[ x = 5 \]

\[ y = 8 \]

Example:

That's not really simple (it tends to do weird stuff)

\[ \text{Trouble} \]

Getting Fancy

What happened?

The inner function returns a function. The argument to print then

\[ \text{prints (5 + 2)} \]

\[ \text{and (eventually) just 7} \]

So we replace the whole expression with

\[ \text{for } x \text{ and } y \]

\[ x = 5 \]

\[ y = 2 \]

\[ (\lambda x : (\lambda y : ((x + y)^x))(5, 2)) \]

A function call such as

Substitution and Formal Parameters

Now what?

Here, we use a common notation for that (due to Church):

\[ \lambda x, y : ((x + y)^x) \]

And if for \( x \)

\[ x = 5\]

\[ y = 2 \]

Then \( x = y = 5 \)

If we just substitute for the first \( x \) as before:

\[ x = 5 \]

\[ y = 8 \]

# or even worse: 4 = 8

\[ \text{print(4)} \]

. . . we get a wrong result (4 instead of 8).

After one substitution, \( x \) isn't around any more to substitute for.

We need something stronger.

Trouble

That is, we replace names by their definitions (values)

\[ x = \begin{cases} 5 & \text{if } x = 5 \\ 2 & \text{if } x = 2 \end{cases} \]

\[ x = x \]

\[ z = x \]

That's not really simple (it tends to do weird stuff)

\[ z = x \]

\[ x = x \]

\[ z = z \]

\[ x = x \]

Let's try to explain the effect of
Environments

• An environment is a mapping from names to values.
• We say that a name is bound to a value in this environment.
• In its simplest form, it consists of a single global environment frame:
  
  Global
  
  abs:
  
  pi:3.1415926
  
  radius: 10
  
  square:
  
  func abs(x)
  
  func square(x)
  
  
  Pre-defined
  
  Imported
  
  Assigned
  
  Assigned
  
  by def
  
  from math import pi
  
  radius = 10
  
  def square(x):
  
  return mul(x, x)

  Last modified: Sat Feb 18 02:57:00 2017 CS61A: Lecture #2 19

Slight Change of Notation

• You’ll be using the Python Tutor from time to time, which uses a somewhat different notation for function values. Might as well get used to it (we’ll explain the “parent=Global” in a later lecture):

  Global
  
  abs:
  
  pi:3.1415926
  
  radius: 10
  
  square:
  
  func abs(x)
  
  func square(x)
  
  Pre-defined
  
  Imported
  
  Assigned
  
  Assigned
  
  by def
  
  from math import pi
  
  radius = 10
  
  def square(x):
  
  return x**2

  Last modified: Sat Feb 18 02:57:00 2017 CS61A: Lecture #2 20

Environments and Evaluation

• Every expression is evaluated in an environment, which supplies the meanings of any names in it.
• Evaluating an expression typically involves first evaluating its subexpressions (the operators and operands of calls, the operands of conventional expressions such as \( x*(y+z) \), . . . ).
• These subexpressions are evaluated in the same environment as the expression that contains them.
• Once their subexpressions (operator + operands) are evaluated, calls to user-defined functions must evaluate the expressions and statements from the definition of those functions.

Evaluating User-Defined Function Calls

• Consider the expression \( \text{square}(\text{mul}(x, x)) \) after executing

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

  Global
  
  mul:
  
  x:-2
  
  square:
  
  func mul(L,R)
  
  func square(x)
  
  square(mul(x,x))

  Evaluation
  
  Environment
  
  Expression
  
  Evaluation

• First evaluate the subexpressions of \( \text{square}(\text{mul}(x, x)) \) in the global environment:

  Global
  
  mul:
  
  x:-2
  
  square:
  
  func mul(L,R)
  
  func square(x)
  
  square(mul(-2,-2))

  For short, just

  mul and square below

  • Evaluating subexpressions \( x, \text{mul}, \text{square} \) take values from the expression’s environment.

Evaluating User-Defined Functions Calls (II)

• Then perform the primitive multiply function:

  Global
  
  mul:
  
  x:-2
  
  square:
  
  func mul(L,R)
  
  func square(x)
  
  square(4)

• In the simplest form, it consists of a single global environment frame:
• We say that a name is bound in a value in this environment.
• An environment is mapped from names to values.
Evaluating User-Defined Functions Calls (IV)

• To explain parameter to user-defined square function, extend environment with a local environment frame, attached to the frame in which square was defined (the global one in this case), and giving x the operand value.

• Now replace original call with evaluating body of square in the new local environment.

Global mul:

\[ x := -2 \]

square:

\[ x := 4 \]

Parent Environment

\[ \text{func} \ mul(L,R) \ [\text{parent} = \text{Global}] \]

\[ \text{func} \ square(x) \ [\text{parent} = \text{Global}] \]

\[ \text{square} \ (4) \]

\[ \text{mul} \ (4, 4) \]

Last modified: Sat Feb 18 02:57:00 2017 CS61A: Lecture #2

Evaluating User-Defined Functions Calls (V)

• When we evaluate \( \text{mul}(x, x) \) in this new environment, we get the same value as before for \( \text{mul} \), but the local value for \( x \).

• When evaluating an identifier in a chain of environments, follow the parent environment links to the first frame containing its definition.

Global mul:

\[ x := -2 \]

square:

\[ x := 4 \]

func mul(L,R) [parent=Global]

func square(x) [parent=Global]

\[ \text{square} \ (4) \]

\[ 16 \]

\[ \text{mul} \ (4, 4) \]

Last modified: Sat Feb 18 02:57:00 2017 CS61A: Lecture #2

So How Does This Help?

• The original problem that led to this whole environment diagram was how to deal with:

\[ x := 4 \]

\[ x := 8 \]

\[ \text{print}(x) \]

• Now it's easy. Each time we assign to x, we create a new binding for it in the current execution frame (replacing the old one, if any).

• When we look up x in the modified environment, we get the new (last assigned) value.

Last modified: Sat Feb 18 02:57:00 2017 CS61A: Lecture #2

Evaluating User-Defined Functions Calls (VI)

• The original problem that led to this whole environment diagram was how to deal with:

\[ x := 4 \]

\[ x := 8 \]

\[ \text{print}(x) \]

• Now it's easy. Each time we assign to x, we create a new binding for it in the current execution frame (replacing the old one, if any).

• When we look up x in the modified environment, we get the new (last assigned) value.

Last modified: Sat Feb 18 02:57:00 2017 CS61A: Lecture #2

Evaluating User-Defined Functions Calls (VII)

• The original problem that led to this whole environment diagram was how to deal with:

\[ x := 4 \]

\[ x := 8 \]

\[ \text{print}(x) \]

• Now it's easy. Each time we assign to x, we create a new binding for it in the current execution frame (replacing the old one, if any).

• When we look up x in the modified environment, we get the new (last assigned) value.

Last modified: Sat Feb 18 02:57:00 2017 CS61A: Lecture #2